

UNLIMITED

92-729

901150
004576

**Automatic Modulation Recognition Based On Spatial
Pattern Classification Using Artificial Neural Networks (U)
FINAL REPORT**

DSS Contract #027.ST.W7714-1-9531/01-ST
July 1992

Author: **Nasir Ghani**
Principal Investigator: Dr. S. Haykin
McMaster University
Communications Research Laboratory
Hamilton, Ontario
L8S 4K1, Canada

Scientific Authority: **René Lamontagne**
Defence Research Establishment Ottawa
Communication Electronic Warfare Section

COPYRIGHT (c) HER MAJESTY THE QUEEN IN RIGHT OF CANADA
AS REPRESENTED BY THE DEPARTMENT OF NATIONAL DEFENCE (1992)

Abstract

This research involves work on an electronic countermeasures contract for the Defense Research Establishment Ottawa (DREO). In this project, an automatic modulation recognition system has been designed and tested based on backpropagation neural networks. This system successfully distinguishes between the specified 10 signal classes to a very high degree and uses a Welch periodogram preprocessor. Experimental results show that the neural networks match and even outdo the performance of the conventional K-nearest neighbour classifier for the same preprocessor, upper 90's percentages. Significant optimization of the neural networks is also shown using the optimal brain damage pruning algorithm.

The design issues, preprocessing techniques, backpropagation algorithms, implementation details, and results of this project are fully discussed in Part I of this document. Part II discusses the specifics of the software developed during the project.

Part I

**Automatic Modulation
Recognition - A Neural
Network Approach**

Contents

1	Introduction	2
1.1	Background	2
1.2	Overview of AMR System	3
1.3	Scope of Thesis	4
1.4	Survey of Chapters	5
2	Input Signals	6
2.1	Signal Specifications	6
2.2	Modulation Simulators	7
2.2.1	Analog Simulators	7
2.2.2	Digital Simulators	9
3	Signal Preprocessing	11
3.1	Regular Periodogram	12
3.2	Welch's Modified Periodogram	16
3.3	Higher Order Statistics	19
3.3.1	Introduction	19
3.3.2	Bispectrum	21

4	Pattern Classifiers	27
4.1	K-Nearest Neighbour Classifier	27
4.2	Backpropagation Neural Network Classifier	30
4.2.1	Summary of Backpropagation Algorithm	31
4.2.2	Using Backpropagation Networks	35
4.2.3	Network Pruning: Optimal Brain Damage	39
4.3	Measuring Classifier Error Rates	43
5	Implementation of Backpropagation Networks	46
5.1	Overview of the WARP Systolic Array	46
5.2	The Batch Backpropagation Algorithm	48
5.3	The Batch Optimal Brain Damage Algorithm	49
6	Results	52
6.1	Preprocessing	53
6.1.1	PSD Averaging	53
6.1.2	Welch's Modified Periodogram	56
6.1.3	Bispectrum	60
6.2	Classifiers	66
6.2.1	K-Nearest Neighbour Classifier	66
6.2.2	Backpropagation Neural Network Classifier	68
6.2.3	Optimal Brain Damage	75
7	Conclusions	86
7.1	Summary	86
7.2	Recommendations	87
A	Overview of Modulation Schemes	89
A.1	Amplitude Modulation	89
A.2	Frequency Modulation	90
A.3	Single Side Band Modulation	91
A.4	Amplitude Shift Keying	92
A.5	Binary Phase Shift Keying	93

A.6	Quadrature Phase Shift Keying	93
A.7	Frequency Shift Keying	94
A.8	Continuous Wave Modulation	95
B	Signal Specifications	96
B.1	Amplitude Modulation	96
B.2	Frequency Modulation	98
B.3	Single Side Band Modulation	99
B.4	Amplitude Shift Keying	100
B.5	Quadrature Phase Shift Keying	101
B.6	Binary Phase Shift Keying	102
B.7	Frequency Shift Keying	103
B.8	Continuous Wave Modulation	105
	Bibliography	106

List of Tables

6.1	Results for 1-NN using PSD averaging for AM/FM problem	55
6.2	Results for 1-NN using periodogram for AM/FM problem	57
6.3	Results for 1-NN using bispectrum for AM/FM problem	62
6.4	Percentage accuracies of 1-NN for 10-classes, 1,000 test per class, B bins extracted	68
6.5	Performance of fully-connected backpropagation network with single hidden layer of H hidden neurons, W weights, 42 inputs (6kHz). Re- sults averaged over 3 independant training runs.	69
6.6	Performance of fully-connected backpropagation network with single hidden layer of H hidden neurons, W weights, 70 inputs (10kHz). Re- sults averaged over 3 independant training runs.	69
6.7	Performance of fully-connected, double-hidden layer backpropagation network, H_1 hidden neurons in first hidden layer, W weights, 30 neu- rons in second hidden layer, 42 inputs. Results averaged over 3 inde- pendant training runs.	73
6.8	Performance of fully-connected, double-hidden layer backpropagation network, H_1 hidden neurons in first hidden layer, W weights, 30 neu- rons in second hidden layer, 70 inputs. Results averaged over 3 inde- pendant training runs.	74
6.9	Sample OBD run for 42 inputs, 120 hidden. W is number of weights, and * indicates results of fully-connected network.	77
6.10	Sample OBD run for 70 inputs, 130 hidden. W is number of weights, and * indicates results of fully-connected network.	78

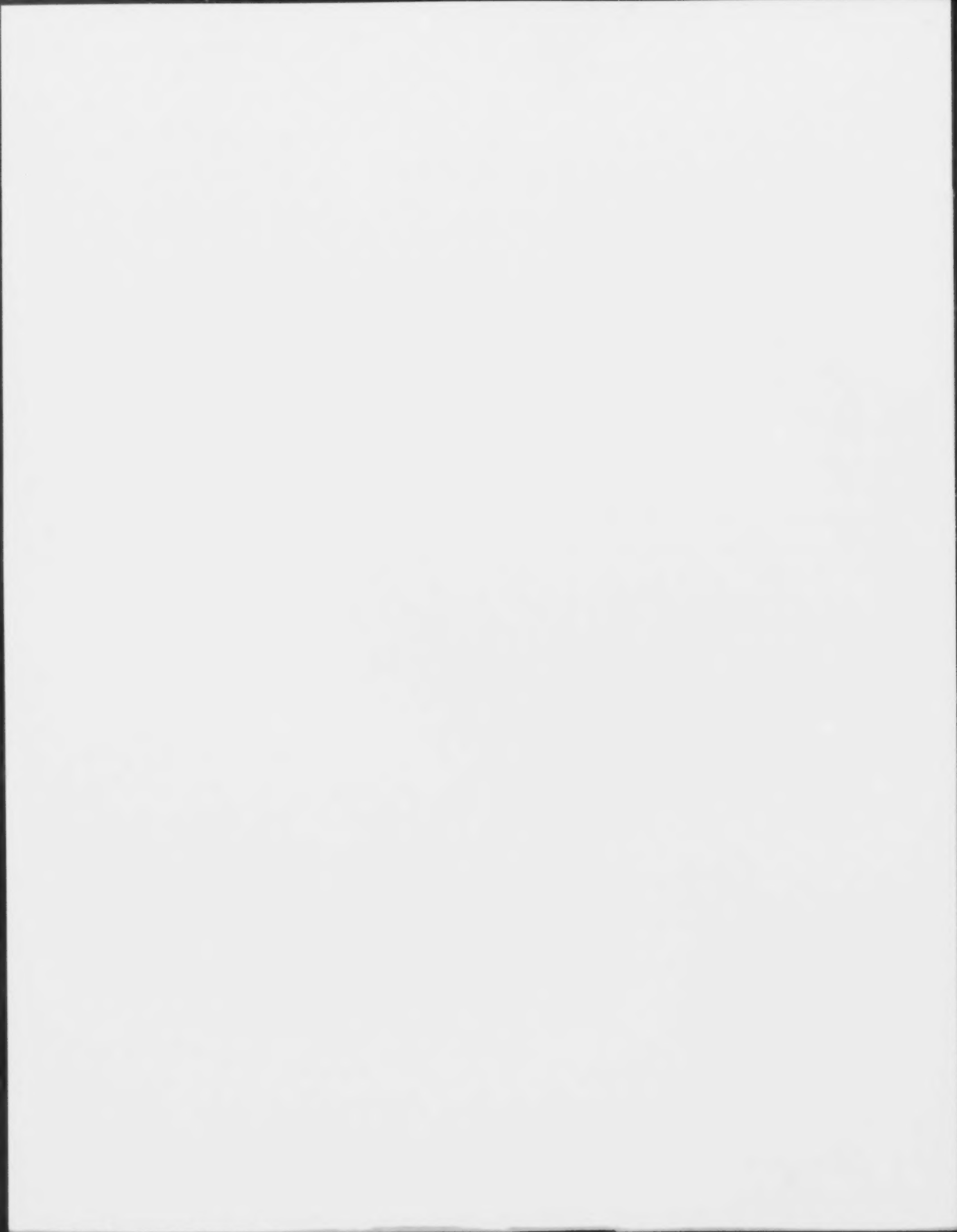
6.11	Sample OBD run for 70 inputs, 120 hidden. W is number of weights, and * indicates results of fully-connected network.	78
6.12	Sample OBD run for 42 inputs, 70 hidden 1, 30 hidden 2. W is the number of weights, and * indicates results of fully-connected network.	84
6.13	Sample OBD run for 70 inputs, 140 hidden 1, 30 hidden 2. W is the number of weights, and * indicates results of fully-connected network.	84

List of Figures

1.1	Overview of Automatic Modulation Recognition System	4
2.1	Overview of Analog Modulation Simulator	8
2.2	Overview of Digital Modulation Simulator	10
3.1	Overview of the PSD averaging preprocessing technique	17
3.2	Overview of Welch averaging method for preprocessing	18
3.3	12-slice bispectrum domain partition. Each slice completely describes the bispectrum	23
3.4	Zoom-in of first discrete bispectral slice.	23
4.1	Fully-connected two-hidden layer backpropagation network	31
4.2	A computing element, neuron, of a backpropagation network	32
4.3	Overtraining in backpropagation networks	39
5.1	Overview of WARP Systolic Array Processor Setup	47
5.2	Description of an iteration of the batch backpropagation algorithm on the WARP systolic array. (Weights and interpolation tables assumed loaded in all cells).	50
6.1	Performance of 1-NN for AM/FM Problem, PSD Averaging	56
6.2	Performance of 1-NN for AM/FM Problem, Welch Periodogram	58
6.3	Welch periodogram for FM signal, $M = 256$, 70 bins (10kHz)	58
6.4	Scaled-log of Welch periodogram for FM signal, $M = 256$, 70 bins (10kHz)	59

6.5	Sample PSD averaged output for AM signal, 80-filter set chosen to represent 10kHz bandwidth.	59
6.6	Sample Welch periodogram output for AM signal, $K = 32$, $M = 256$, 70 bins chosen to represent 10kHz bandwidth	60
6.7	Sample bispectrum of an AM signal, FFT size, $M = 64$, 272 points overall	61
6.8	Sample bispectrum of an FM signal, FFT size, $M = 64$, 272 points overall	62
6.9	Plot of bispectral parameter count vs. FFT size, M	63
6.10	Averaged Welch Periodograms, $M = 256$. AM (left), FM (right) . . .	64
6.11	Averaged Welch Periodograms, $M = 256$. USB (left), LSB (right) . .	64
6.12	Averaged Welch Periodograms, $M = 256$. ASK (left), BSPK (right) .	65
6.13	Averaged Welch Periodograms, $M = 256$. QPSK (left), slow-rate FSK (right)	65
6.14	Averaged Welch Periodograms, $M = 256$. fast-rate FSK (left), CW (right)	66
6.15	Performance of k-NN for Welch Periodogram, $M = 256$, 10 classes .	67
6.16	Fully-connected single hidden layer network performance, Welch periodogram, $M = 256$	70
6.17	Fully-connected double hidden layer network performance for 42 inputs, Welch periodogram, $M = 256$	72
6.18	Fully-connected double hidden layer network performance for 70 inputs, Welch periodogram, $M = 256$	73
6.19	1000-bin histogram plot of an AM signal	76
6.20	1000-bin histogram plot of an FM signal	76
6.21	Kurtosis vs. skew plot, AM and FM classes, 300 samples each	77
6.22	OBD for 42 inputs, 120 hidden. Fully connected network has 6,370 weights.	79
6.23	OBD for 70 inputs, 130 hidden. Fully connected network has 10,540 weights.	79

6.24	OBD for 70 inputs, 120 hidden. Fully connected network has 9,730 weights.	80
6.25	Cumulative weight magnitude vs. weight, 70 inputs, 120 hidden, 6331 weights (ie. 35% deletion).	80
6.26	Cumulative weight magnitude vs. weight, 70 inputs, 130 hidden, 3786 weights (ie. 64% deletion).	81
6.27	Cumulative Saliency Distribution Plot, 70 inputs, 130 hidden, 500 histogram bins used	83
6.28	OBD for 42 inputs, 70 hidden 1, and 30 hidden 2. Fully connected network has 5,450 weights.	85
6.29	OBD for 70 inputs, 140 hidden 1, and 30 hidden 2. Fully connected network has 14,480 weights.	85



Contents

Chapter 1

Introduction

1.1 Background

Ever since the dawn of radio communication, there has been a need to monitor electromagnetic signal transmissions in the RF spectrum. This requirement has grown with the rapid advance of technology and society's increasing demand for radio communications. Civilian authorities need to monitor RF activity over their territory to enforce legal transmission guidelines. Military authorities, on the other hand, must monitor RF activity for reasons of national security.

The most common technique used to monitor RF activity is human operator-based. In this scheme, an operator scans the RF spectrum with a wide-band receiver and visually checks for signal transmissions on some sort of visual display unit. There can be many variations to this technique, using multiple receivers and/or operators. For example, one operator (master) can scan the complete desired range in the RF spectrum. When a transmission is detected, the signal's location is passed to a second operator (slave) who then uses a finer receiver tuned close to the frequency of the intercepted signal to further observe the signal. However, these techniques have a problem in that they are operator based. This makes the performance of the system slow and dependant on the experience and ability of the operator(s). The extremely dense radio spectrum compounds this problem, with more and more signals being present in a given bandwidth. It is therefore highly desirable to have an automated

system which can detect, demodulate and extract various signal attributes from transmitted signals. Such a system can be used as a tool by human operators, allowing them to perform higher level functions and thereby monitoring the RF spectrum more effectively.

The Communication Electronic Warfare Section (CEW) at the Defense Research Establishment Ottawa (DREO) has been looking into developing an automated monitoring system. Such a system should be able to perform the required tasks faster and more accurately than the present operator-based systems. A major attribute of an intercepted signal to be determined is its modulation type. For example, it is crucial to know the modulation format if signal is to be decoded. As a result, a modulation recognition subsystem is required, one which takes an inputted signal and determines its modulation type among a set of possible ones. Such an automatic modulation recognition (AMR) system is essentially a pattern recognition system which classifies intercepted signals as belonging to one of several classes. System performance is judged by the accuracy of the system in correctly identifying the modulation type and its speed.

1.2 Overview of AMR System

Figure 1.1 presents an overview of a proposed AMR system. The first stage is a preprocessor which takes a sampled signal and preprocesses it to remove noise and perform feature extraction. It is assumed that the input signal is sampled at over twice the maximum bandwidth the system is to handle (Nyquist criteria). The feature extraction process is crucial, requiring the dimensionality of the feature space to be reduced as much as possible without undermining system performance. Large feature sets can have unrealistic computation requirements which may make implementation very slow and difficult. The second stage is a classifier which uses the preprocessed features to compute further outputs. This classifier may either be a conventional type (ie. K-nearest neighbour) or a neural-network. The last stage is an output processor (decision device) which takes the outputs of the classifier and gives the best estimate of the modulation class.

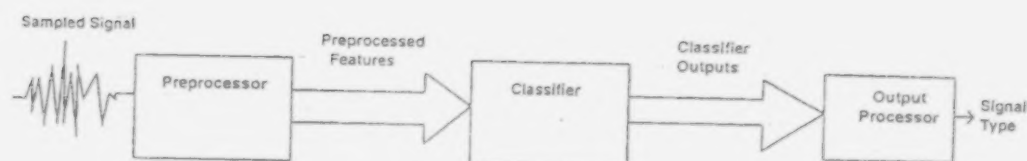


Figure 1.1: Overview of Automatic Modulation Recognition System

1.3 Scope of Thesis

The purpose of this project is to investigate the performance of several preprocessors and backpropagation neural network classifiers for the AMR problem. The performance of these networks is to be compared against that of K-nearest neighbour classifiers. Specifically, the system is to handle 10 modulation types: amplitude modulation (AM), frequency modulation (FM), single side band upper/lower (SSBU/SSBL), amplitude shift keying (ASK, ie. OOK), bipolar phase shift keying (BPSK), quadrature phase shift keying (QPSK), frequency shift keying (slow rates, FSK1, and fast rates, FSK2), and continuous wave modulation (CW, "no-information" class). The last class, CW, is added to to prevent false detection. The various modulation parameters of these signals are specified by the DREO [5], and all signals are generated by simulators (provided by the DREO). The performance of spectral preprocessing techniques is to be investigated. The size of the problem requires the use of the WARP systolic array processor to train/test the neural network classifiers. In addition, it is desired to optimize the neural networks to make the design more efficient and easier

to implement.

1.4 Survey of Chapters

Chapter 2 discusses some of the details regarding the signals and presents overviews of the signal modulators.

In Chapter 3, various preprocessing paradigms are described. In particular several power spectrum estimation techniques are described along with some higher order techniques.

Chapter 4 discusses the classifiers to be used. In particular, the conventional K-nearest neighbour classifier is described along with the widely used backpropagation network. The optimal brain damage technique for pruning backpropagation networks is summarized and details given about estimating classifier error rates.

Chapter 5 summarizes the implementation of the regular and pruned backpropagation networks on the WARP parallel systolic array. In addition, an overview of the WARP is presented.

Detailed results from the experiments done with the preprocessors, neural networks, and K-nearest neighbour classifiers are presented in Chapter 6. Comparisons for networks of different sizes are given and results from the pruned networks are also presented.

Chapter 7 draws conclusions based on the experimental results. Some recommendations for future work are also given. The Appendices review all the modulation techniques and summarize the DREO specifications for each one.

Chapter 2

Input Signals

Simulators are used to generate the input signals to the AMR system. These simulators are written in *C* and run in the *SUN Unix* environment. DREO specifications explicitly state details about all signal classes for the AMR system which are used to further define the classes of signals. These specifications are required to derive inputs to the simulators to generate the samples. This chapter addresses various issues regarding the signals and goes on to describe the functionality of the simulators. An overview of each modulation type and corresponding details from the DREO specifications are given in the Appendices.

2.1 Signal Specifications

The system sampling rate is set to 75kHz. In other words, all signals are generated at this, and this rate can theoretically allow signals of upto 37.5kHz (Nyquist criterion). This is a very high rate and is well over twice the maximum bandwidth the system is to handle, given in the DREO specifications as 25kHz. The specifications require a sampling rate between 60 and 100kHz, and a sample time of between 100 and 200ms [5]. The sample time is chosen to be 110ms. There are several reasons for these two choices. First, the higher sampling rate will make a possible hardware implementation easier, since it more than satisfies the Nyquist criteria for even the larger bandwidth signals. Second, this duration yields 8250 samples, which is close to a power of 2,

2^{13} , and will allow the use of the Fast Fourier Transform (FFT) for faster spectral preprocessing. In addition, some of the slower digital modulations have data rates as low as 50 baud, and using smaller time durations may give a lot of signals with no bit transitions at all. This will not give a good time representation of the signal, making the noise appear as the only random quantity. Specifically, 110ms gives about 5 bit transitions for the slowest ASK signal (45 baud). Therefore the highest possible percentage probability of having all 0's or all 1's in a 110ms timespan is $\approx 6.25\%$, which is deemed as an acceptably low value.

2.2 Modulation Simulators

There are two major classes of simulators, one for the analog modulations and another for the digital. High-level descriptions of these simulators are now given.

2.2.1 Analog Simulators

There are three analog modulation simulators, one pertaining to each class: AM, FM, and SSB (upper and lower). Figure 2.1 shows an overview of the analog modulation simulator. The input to the system is a voice signal of a certain duration, sampled at 16kHz. Such signal segments are obtained from one of 15 voice files containing large duration samples of English speech from various speakers. The exact file and the corresponding starting location for a speech segment are both selected randomly. Once obtained, the voice signal is filtered to ensure that its spectral content lies between 300Hz and 3000Hz. Generally, since the exact bandwidth of an incoming signal is unknown, a high *system sampling rate* is chosen to generate the samples. Thus, after the filtering operation, the voice signals have to be interpolated (ie. upsampled) to the desired system sampling rate, 75kHz. After upsampling, the signal is modulated by one of the analog techniques to an intermediate frequency (IF), mean chosen to be 15kHz. The standard deviation of the IF is set to 10% of the signal's bandwidth to account for error in IF-estimation. The subsequent operation again filters the signal, ensuring that the transmitted bandwidth is not larger than a certain pre-set range

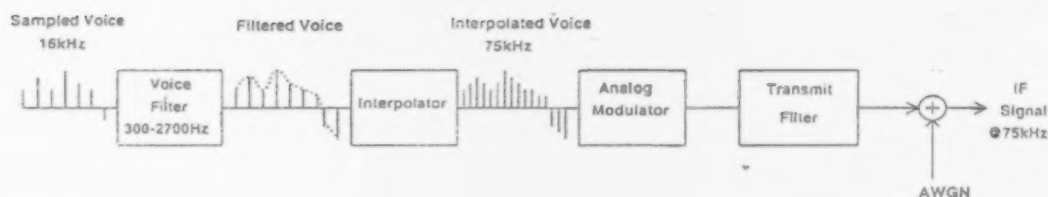


Figure 2.1: Overview of Analog Modulation Simulator

(like transmit filter). The bandwidth of this filter is dependant upon the modulation technique being used and is given in the specifications. AM signals are limited to a 6kHz bandwidth, FM to 25kHz and SSB transmissions to 2.7kHz. In the final step, additive white Gaussian noise (AWGN) is added to the signal to generate the output, an analog modulated voice signal at a given IF. Specifications state SNR values ranging from 5dB to 25dB. Specifically, the SNR is chosen by a "truncated" Gaussian distribution, with 11dB mean and 6dB standard deviation. Any random value generated outside the 5dB-25dB range is forced to the the closest edge value. Such a formulation is biased, generating more values closer to the lower bound of the SNR range. This is the intent of the design, to implement the AMR system with more noisy signals. All voice and transmit filters are finite impulse response (FIR) filters, generated by smoothing a a truncated and delayed ideal impulse response with a Kaiser window. The interested reader is referred to the references [11], [9] for detailed descriptions of these filters and their implementation.

2.2.2 Digital Simulators

There are a total of five different digital modulation simulators: ASK, BPSK, QPSK, and FSK. Figure 2.2 shows the overview of these simulators. The input to the system is a binary bitstream, read from one of 10 stored digit files. These files are created by implementing a psuedo-random generator with a feedback shift register [11]. The file number and corresponding start location of the bitstream are selected randomly, as is done with the voice files in the analog simulators. This bitstream is upsampled to the system sampling rate and passed on to the appropriate digital modulator. The IF output of the modulator is filtered to ensure that it meets transmission bandwidth requirements and then noise is added (same SNR's as in analog case).

DREO specifications state data rates for all modulation types and hence the transmit filter bandwidths can be computed. The details of these computations can be found in Appendix B, with the results being summarized as follows: ASK is bandlimited to 3kHz, FSK1 to 6kHz, and FSK2 to 25kHz. Due to the fact that the remaining digital modulations can only transmit at groups of certain fixed rates, the bandwidth of the transmit filter is data-rate dependant. BPSK is given a bandwidth 6 times the data rate and QPSK 5 times the data rate. As in the analog case, the filters are implemented using a Kaiser windows. Also, the mean IF is the same as the analog value, 15kHz, and the standard deviation set to 10% of the signal's bandwidth.

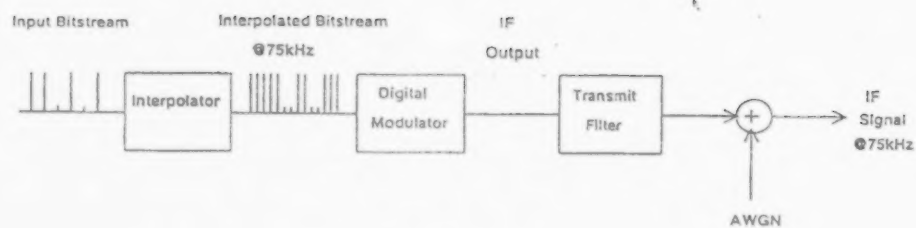


Figure 2.2: Overview of Digital Modulation Simulator

Chapter 3

Signal Preprocessing

Signal preprocessing performs feature extraction on the input data. A major reason for this step is to map the initial feature space (time) into a more effective one [2] in which the classes are well separated. Generally a lot of discriminatory information is "hidden" in the time samples and therefore techniques have to be developed to extract it. By effective it is meant features which give added separability between the various modulation classes. Many theoretical separability measures exist but most assume knowledge of class-conditional probability distributions (ie. parametric estimation required). In addition, even if the exact pdf form and parameter(s) are known, minimizing a cost function based on a given separability measure to determine a feature set is computationally prohibitive. As a result, it is decided to investigate the performance of various techniques which are postulated to perform well. Since most cost functions are difficult to formulate without assuming some underlying distribution of the data, a simple yet practical measure of a given feature set is simply the error count on a large, independent test set.

Another reason for performing preprocessing is to reduce the dimensionality of the initial feature set. Smaller input dimensionalities give a considerable reduction in the training and test set sizes required to adequately train a classifier. In addition, the classifier training time is also smaller for more compact feature sets. An even more serious problem lies in choosing too large a feature set. "Typically, as one increases the number of features extracted from a given training data set, the performance

initially increases but then begins to deteriorate" [3]. This occurs due to the finite training size. As the dimensionality of the feature space increases, the training data set is more sparsely distributed and therefore the probability distributions represented by the sampling are less representative of the true shapes. This is called the "curse of dimensionality problem".

Specifically, it is desired to investigate the performance of some standard spectral techniques along with some incorporating higher order spectra (HOS). It is postulated that the additional information given by HOS will improve the classification between difficult classes which do not perform well with standard techniques. The standard methods chosen are the straight periodogram and the modified Welch periodogram. The higher order method chosen is the bispectrum. Each one of these methods is used to generate features and subsequently design classifiers with. It should be mentioned that *all* preprocessing is done in *C* on *SUN Unix* workstations. The preprocessing techniques are now described in detail.

3.1 Regular Periodogram

Consider a power signal, $x(t)$, whose average power is given by [4]:

$$P = \lim_{T \rightarrow \infty} \frac{1}{T} \int_{-T}^T x^2(t) dt \quad (3.1)$$

Since the energy in $x(t)$ is infinite, its Fourier transform does not exist, and therefore a segment of $x(t)$ has to be taken ($-T \leq t \leq T$) to get some sort of frequency representation. Thus the Fourier transform of the truncated signal (ie. rectangular window) is defined as:

$$X(f) = \int_{-T}^T x(t) e^{-j2\pi ft} dt \quad (3.2)$$

The power spectral density is then defined as being:

$$S_x(f) = \lim_{T \rightarrow \infty} \frac{1}{2T} |X(f)|^2 \quad (3.3)$$

The expression $\frac{1}{2} |X(f)|^2$ is called the *periodogram* of $x(t)$. This quantity is dependant on the duration of the segment taken, $2T$, and the longer this value, the closer

it is to the true power spectral density. Note that this quantity does not contain any phase information about the power spectrum. This can give problems discriminating between signals which have similar power spectra but different phase spectra (digital modulations). The power spectral density is also defined as the Fourier transform of the autocorrelation of a signal:

$$S_x(f) = \int_{-\infty}^{+\infty} R_x(\tau) e^{-j2\pi f\tau} d\tau \quad (3.4)$$

where

$$R_x(\tau) = \lim_{T \rightarrow \infty} \frac{1}{2T} \int_{-T}^{+T} x(t)x(t-\tau)dt \quad (3.5)$$

Now consider the sequence of samples of $x(t)$, $\{x_n\}$ for $0 \leq n \leq N-1$. The Fourier transform of this sequence is defined as:

$$X(e^{j\omega}) = \sum_{n=0}^{N-1} x(n) e^{-j\omega n} \quad (3.6)$$

from which the autocorrelation and power spectral density are:

$$\phi_{xx}(m) = E[x_n x_{n+m}] \quad (3.7)$$

$$P_{xx}(\omega) = \sum_{n=0}^{N-1} \phi_{xx}(n) e^{-j\omega n} \quad (3.8)$$

Using Eq. 3.3 and 3.6, the periodogram of the sequence is simply [12]:

$$I_N(\omega) = \frac{1}{N} |X(e^{j\omega})|^2 \quad (3.9)$$

It can be shown [12] that due to the finite length of the sequence, $I_N(\omega)$ is a biased (asymptotically unbiased) estimate of the true power spectrum, $P_{xx}(\omega)$. The mean of I_N is given by:

$$E[I_N(\omega)] = \sum_{m=-(N-1)}^{N-1} \left(\frac{N-|m|}{N} \right) \phi_{xx}(m) e^{-j\omega m} = \frac{1}{2\pi} \int_{-\pi}^{+\pi} P_{xx}(\theta) W_B(e^{j(\omega-\theta)}) d\theta \quad (3.10)$$

where $P_{xx}(\theta)$ is the true spectrum and

$$W_B(e^{j\omega}) = \frac{\sin(\frac{\omega(2N-1)}{2})}{\sin(\frac{\omega}{2})} \quad (3.11)$$

(ie. estimate is windowed in time by triangular window and therefore convoluted in frequency by its Fourier transform). It is obvious that as $N \rightarrow \infty$, the bias in the estimate tends to zero. A lot of tedious manipulations show that for any nonwhite Gaussian process, the variance of the estimate $I_N(\omega)$ is:

$$\sigma_{I_N(\omega)}^2 \cong P_{xx}^2(\omega) \left(1 + \left(\frac{\sin(\omega N)}{N \sin(\omega)} \right)^2 \right) \quad (3.12)$$

where σ_x^2 is the variance of the sequence. Since this value is never zero, it means that the estimate is not *consistent* (ie. variance does not tend to zero as $N \rightarrow \infty$). Non-Gaussian signals contain additional terms in the variance expression, proportional to $\frac{1}{N}$, so the inconsistent nature holds for this case also.

For frequencies $\omega_1 = \frac{2\pi k}{N}$ and $\omega_2 = \frac{2\pi l}{N}$, where $k \neq l$, it can be shown that the values of $I_N(\omega)$ are uncorrelated [12]. This fact is important since the discrete Fourier transforms are used to estimate the periodogram. It means that there will be increased fluctuation between successive spectral samples spaced closer together (due to zero covariance between them). This is a unfortunate, considering that good estimators should give more and more "consistent" estimates of the spectrum as the sample size N increases. Hence increasing N reduces the bias but leads to increased variability in the estimate.

A way to reduce the variability of the estimated spectrum is to apply a time window to the samples before computing the Fourier transform. Ideally, this means that the true spectrum of the signal is convoluted with the Fourier transform of the windowing function, $W(e^{j\omega})$:

$$S_{xx}(\omega) = \frac{1}{2} \int_{-\pi}^{+\pi} P_{xx}(\theta) W(e^{j(\omega-\theta)}) d\theta \quad (3.13)$$

However, since the practical method being implemented to compute the estimates is biased, further windowing means that the spectrum estimate is now masked by *two* windows—the triangular one, introduced by the bias, and the applied one. In the frequency domain, this results in a "double" convolution of the true spectrum with the windows' spectra. In other words, the mean of the estimate now becomes

$$E[S_{xx}(\omega)] = \frac{1}{2\pi} \int_{-\pi}^{+\pi} E[I_N(\theta)] W(e^{j(\omega-\theta)}) d\theta \quad (3.14)$$

where $E[I_N(\omega)]$ is given by Eq. 3.10 and $W(e^{j\omega})$ is the Fourier transform of the applied window.

If it is assumed that $W(e^{j\omega})$ is wide compared to $W_B(e^{j\omega})$ (ie. $M \ll N$), an approximate expression for the variance of $S_{xx}(\omega)$ can be derived for nonwhite Gaussian signals. A lot of tedious manipulations show that the variance of $S_{xx}(\omega)$ differs from that of $I_N(\omega)$ only by a constant factor and is given by [12]:

$$\sigma_{S_{xx}}^2(\omega) = \frac{1}{N} \sum_{-(M-1)}^{M-1} w^2(m) \sigma_{I_N}^2(\omega) \quad (3.15)$$

where $2M + 1$ is the length of the window. Hence by choosing the window shape and size correctly, the variance of the estimate can be reduced. There are many windows which can be used, and the choice depends upon the application. For example, Hamming, Hanning, and Blackmann-Harris windows all give less sideband ripple (reduced Gibb's phenomena) at the expense of a wider mainlobe, smoothing sharp spectral-transitions [12]. Specifically, the Hamming window is chosen to be used:

$$w(n) = \begin{cases} 0.54 - 0.46 \cos\left(\frac{2\pi n}{M}\right), & 0 \leq n \leq M \\ 0, & \text{otherwise} \end{cases} \quad (3.16)$$

The scaling factor for the variance can be computed and for $N = M = 8192$, and is 0.3974. This means that there is a reduction of roughly 2.5 times in the variance of the estimate of the power spectral density using the Hamming window over the rectangular (unsmoothed) window.

From Eq. 3.9, it is seen that the periodogram estimate maps N points in the sample domain to N points in the spectral domain (representing frequencies up to $\frac{f_s}{2} = 37.5\text{kHz}$). However, since N is very large (8192), using all spectral bins as input features to the classifier is impractical. The large value of N occurs due to the high sampling rate and long sample time chosen to adequately observe all signals. A simple way to reduce the input dimensionality is to average groups of bins together to obtain a coarser estimate of the spectrum. There is an obvious tradeoff here: trading spectral resolution for more compact feature sets. Reduced resolution generally reduces the discriminatory power of a feature set. By overlapping the values averaged, additional correlation can be introduced between successive features giving a smoother

estimate. Apriori information can also be used in processing the raw periodogram to significantly reduce input dimensionality. For example, if all signals are known to lie within a certain bandwidth (25kHz), then only the spectral samples corresponding to this range need be extracted. Furthermore, since it is known that most of the discriminatory information lies close to the mean IF (ie. centre of bandwidth), smaller averages can be taken near the mean IF and larger ones close to the edge of bandwidth. This will result in better resolution for regions with higher discriminatory information. This periodogram lumping technique is dubbed "PSD averaging" (also, please see Figure 3.1 for a summary of this technique). Extensive experimentation is required to determine "good" feature sets using the PSD averaging. A benefit of this approach is the fact that the maximum resolution in the frequency domain is very good ($\frac{2f}{N} = 9.155\text{Hz}$). This is a direct consequence of the large N , and allows the technique to give any resolution which is a multiple of the resolution value. As previously mentioned, an obvious drawback of the technique is the high variability of the estimate (despite windowing). The fact that the signals have a stochastic nature makes it even more important to take steps to reduce the variability in the estimates. A standard approach to reducing the variance of estimates is to average over a number of independent estimates [12]. This leads to the second preprocessing method, the modified periodogram based on Welch's method.

3.2 Welch's Modified Periodogram

Welch has formulated another approach to estimating the spectrum of a given sequence. This method uses both windowing and averaging to reduce the variability in the estimates. However, as it will be shown, these improvements are not without cost.

Consider again N sample points, $\{x_n\}$ of a signal $x(t)$. Assuming $N = KM$, one can divide this sequence into K segments of M points and compute windowed periodogram estimates as follows:

$$x^i(n) = x(n + iM), 0 \leq n \leq M - 1, 0 \leq i \leq K - 1 \quad (3.17)$$

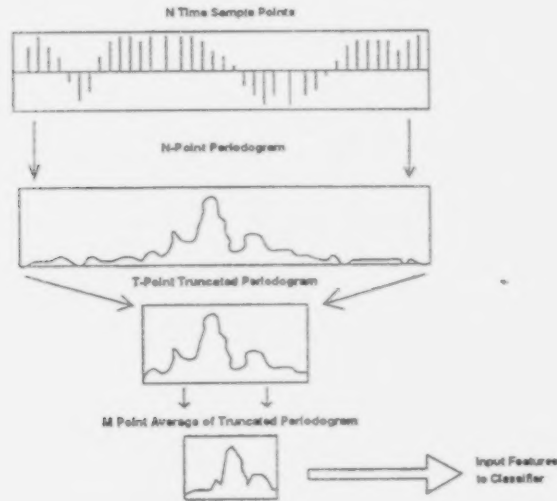


Figure 3.1: Overview of the PSD averaging preprocessing technique

$$J_M^i(\omega) = |\mathcal{F}\{x^i(k)w(k)\}|^2 = \frac{1}{MU} \left| \sum_{n=0}^{M-1} x^i(n)w(n)e^{-j\omega n} \right|^2 \quad (3.18)$$

where

$$U = \frac{1}{M} \sum_{n=0}^{M-1} w^2(n) \quad (3.19)$$

and $w(n)$ is a chosen window. The spectrum estimate is therefore given by a average of the K M -point periodograms, $J_M^i(\omega)$:

$$S_{xx}(\omega) = \frac{1}{K} \sum_{i=0}^{K-1} J_M^i(\omega) \quad (3.20)$$

The mean of this estimate is there given by [12, p.553]:

$$E[S_{xx}(\omega)] = \frac{1}{2\pi} \int_{-\pi}^{+\pi} P_{xx}(\theta) W(e^{j(\omega-\theta)}) d\theta \quad (3.21)$$

where $W(e^{j\omega})$ is the Fourier transform of the truncated, M point window, $w(n)$:

$$W(e^{j\omega}) = \frac{1}{MU} \left| \sum_{n=0}^{M-1} w(n)e^{-j\omega n} \right|^2 \quad (3.22)$$

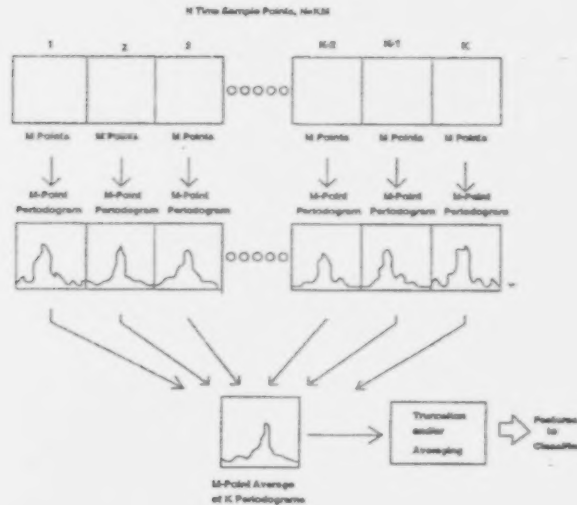


Figure 3.2: Overview of Welch averaging method for preprocessing

(Figure 3.2 presents a visual description of the method). Note that now window sizes are of length M and not N , and hence their spectrum is significantly wider, and the resolution correspondingly reduced. Welch has shown [14] that the variance of his estimation method for a Gaussian process using non-overlapped windows is:

$$\sigma_{S_{xx}(\omega)}^2 \approx \frac{1}{K} P_{xx}^2(\omega) \quad (3.23)$$

Welch goes on to show that for overlapped windowing, the above expression gets scaled by a constant factor less than unity, thereby reducing variability further. An obvious drawback in using the Welch method is the fact that the resolution in the frequency domain goes down from $\frac{2f_s}{N}$ Hz to $\frac{2f_s}{M}$ Hz, where $N = MK$. This loss of “fine” detail in the features may pose problems for classes which are very close. However, the variance in the estimate is reduced by a factor of about M , and thus there is a tradeoff here. Choosing M and K for a given N controls the bias and variance respectively, and a lot of experimentation with different values is required to find the “best” values for a given problem. As is the case with the periodogram, apriori information can also be used to further trim the input feature set. Not only can a

specified bandwidth be extracted, but averaging can also be done (at the expense of resolution). However, depending upon the M value chosen, averaging of neighbouring bins may not be required.

3.3 Higher Order Statistics

3.3.1 Introduction

The above preprocessing methods rely on using the power spectrum to generate features for the classifier. Since the information contained in the power spectrum is essentially that which is present in the autocorrelation, this would suffice for complete statistical description of a Gaussian process [15]. However, if the signals are not Gaussian, it may be worthwhile to characterize deviation from Gaussianity. It is here that higher order spectra can be used. The bispectrum is a higher order spectral technique and is chosen for the preprocessing. In addition, the higher order moments used in the bispectrum should extract more information about the phase of the time samples.

Consider a single Gaussian random variable, x , whose characteristic function is defined as:

$$\phi(\omega) = E[e^{-j\omega x}] = \int_{-\infty}^{+\infty} p(x)e^{-j\omega x} dx \quad (3.24)$$

where $p(x)$ is the probability distribution function (pdf). The r -th order moment is then defined as:

$$m_r = (-j)^r \frac{\delta^r}{\delta \omega^r} \phi(\omega) |_{\omega=0} \quad (3.25)$$

Now for a Gaussian process with μ mean and σ^2 variance, the characteristic function is given by:

$$\phi(\omega) = \int_{-\infty}^{+\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} e^{-j\omega x} dx = \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{+\infty} e^{-\frac{(x-\mu)^2}{2\sigma^2} - j2\sigma^2\omega x} dx = F(\omega, \mu, \sigma) \quad (3.26)$$

Therefore since this is solely a function of ω , $\mu = m_1$, and $\sigma = \sqrt{m_2}$, all moments of order ≥ 3 can be characterized by the first two moments by using Eq. 3.25. Therefore it can be said that there is no additional information being captured by computing

higher order moments for Gaussian random variables. Extending the definition of a characteristic function to multiple random variables ($x_1, x_2, \dots, x_n$), the joint characteristic and moment functions are:

$$\phi(\omega_1, \omega_2, \dots, \omega_n) = E \left\{ e^{j(\omega_1 x_1 + \omega_2 x_2 + \dots + \omega_n x_n)} \right\} \quad (3.27)$$

and

$$m_{k_1 k_2 \dots k_n} = E \left\{ x_1^{k_1} x_2^{k_2} \dots x_n^{k_n} \right\} = (-j)^r \frac{\delta^r \phi(\omega_1, \omega_2, \dots, \omega_n)}{\delta \omega_1^{k_1} \delta \omega_2^{k_2} \dots \delta \omega_n^{k_n}} \Big|_{\omega_1 = \omega_2 = \dots = \omega_n = 0} \quad (3.28)$$

where $r = k_1 + k_2 + \dots + k_n$. In addition, a cumulant function can be defined which explicitly measures the "Gaussianity" of the input data, by giving zero for ideal Gaussian input data and orders greater than two (ie. $k_i \geq 3$). The r -th order cumulant for a single continuous random variable, x is defined as:

$$c_r = (-j)^r \frac{\delta^r \ln \phi(\omega)}{\delta \omega^r} \Big|_{\omega=0} \quad (3.29)$$

and the r -th order cumulant for multiple random variables is given by:

$$c_{k_1 k_2 \dots k_n} = (-j)^r \frac{\delta^r \ln \phi(\omega_1, \omega_2, \dots, \omega_n)}{\delta \omega_1^{k_1} \delta \omega_2^{k_2} \dots \delta \omega_n^{k_n}} \Big|_{\omega_1 = \omega_2 = \dots = \omega_n = 0} \quad (3.30)$$

Using the r -th order cumulant ($k_i = 1 \forall i$), the r -th order spectrum is defined as the Fourier transform of the sequence. Now using the above generalized expressions, it can be seen that the mean and autocorrelation (first and second order statistics) are just special cases of the moment and/or cumulant sequence. For example, letting $r = k_1 = 1$ and $x_1 = X(k)$, (stationary, zero mean) the mean is simply:

$$m_{k_1} = m_1 = c_1 = E \{ X(k) \} \quad (3.31)$$

Now define another variable, $x_2 = X(k+n)$, and $k_2 = 1, r = 2$: the autocorrelation is simply equal to the second order moment/cumulant:

$$m_2(n) = c_2(n) = E \{ X(k) X(k+n) \} \quad (3.32)$$

The 1-dimensional Fourier transform (ie. $r = 1$) of the above moment sequence is equivalent to the second-order spectra of the data, namely the power spectrum. It

can be shown that even upto $n = 3$, the moment and cumulant functions are equal for zero-mean, stationary processes. The second order cumulants are incorporated in the use of the periodogram methods. Therefore to make use of the higher order statistics, Fourier transforms of cumulant sequences with $r \geq 3$ have to be used. It is hoped that these methods give added discriminatory power, especially if the data is non-Gaussian. However, it must be stressed that the increase in data dimensionality is huge with these higher order methods. As a result, to keep the dimensionalities within the bounds of computation, only third-order spectrum (bispectrum) is chosen to be investigated.

3.3.2 Bispectrum

For a stationary random process with zero mean, the bispectrum is defined as the Fourier transform of the third-order moment or cumulant sequence [15]:

$$B(\omega_1, \omega_2) = \mathcal{F} \{E[X(k)X(k+m)X(k+n)]\} = \sum_{m=-\infty}^{+\infty} \sum_{n=-\infty}^{+\infty} R(m, n) e^{-j(\omega_1 m + \omega_2 n)} \quad (3.33)$$

where $R(m, n)$ is the third order moment sequence, defined by extending the derivation for the autocorrelation sequence (Eq. 3.32):

$$R(m, n) = E \{X(k)X(k+m)X(k+n)\} \quad (3.34)$$

Eq. 3.33 hints at a straightforward way to compute estimates of the discrete bispectrum of a given random sequence, $X(k)$. An estimate of the moment sequence, $\hat{R}(m, n)$ can be made and then a two-dimensional Fourier transform can be used to obtain an estimate of the bispectrum. Just as is the case with the power spectrum, there are two equivalent methods of computing the bispectrum. The above method estimates the moments in the time domain and then computes a Fourier transform. Another directly uses Fourier techniques, and since the FFT is available to speed up the computation of the discrete Fourier transform (DFT) of a given sequence, such a technique would be more attractive. Nikias [15] states an equivalent method of computing $B(\omega_1, \omega_2)$ for which uses Fourier transforms of the sequence directly. Let

$$X(k) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} Z(\omega) e^{j\omega k} d\omega \quad (3.35)$$

then

$$E\{Z(\omega_1)Z(\omega_2)Z(\omega_3)\} = \begin{cases} B(\omega_1, \omega_2), & \omega_1 + \omega_2 = \omega_3 \\ 0, & \omega_1 + \omega_2 \neq \omega_3 \end{cases} \quad (3.36)$$

It can also be shown [16, p.873] that the following symmetry properties hold for third moments:

$$\begin{aligned} R(m, n) &= R(n, m) &= R(-n, m - n) \\ &= R(n - m, -m) &= R(m - n, -n) \\ &= R(-m, n - m) \end{aligned} \quad (3.37)$$

Then for real data these symmetries induce the following symmetries in the bispectrum:

$$\begin{aligned} B(\omega_1, \omega_2) &= B(\omega_2, \omega_1) &= B^*(-\omega_2, -\omega_1) \\ &= B^*(-\omega_1, -\omega_2) &= B(-\omega_1 - \omega_2, \omega_2) \\ &= B(\omega_1, -\omega_1 - \omega_2) &= B(-\omega_1 - \omega_2, \omega_1) \\ &= B(\omega_2, -\omega_1 - \omega_2) \end{aligned} \quad (3.38)$$

It can be further shown that the above properties divide the $\omega_1\omega_2$ -plane into twelve *self-contained* triangular regions, where knowledge of the bispectrum in any one of these regions is sufficient to completely describe the bispectrum for all ω_1 and ω_2 (Please see Figure 3.3). Choosing the first triangle in the first quadrant, and using the fact that the bispectrum is periodic with period 2π , one can completely specify the discrete bispectrum within the triangular slice shown in Figure 3.4.

Eq. 3.36 suggests another simple method of estimating the discrete bispectrum of a given random sequence. Given a sequence of N points, $X(k)$, $0 \leq k \leq N - 1$, compute its N -point FFT, $Z(k)$, and then compute a minimal discrete bispectral slice (Figure 3.4) with no averaging as follows:

$$\hat{B}(m, n) = \begin{cases} Z(m)Z(n)Z^*(m+n), & \text{where } \omega_m = \frac{m}{2\pi N}, \omega_n = \frac{n}{2\pi N} \text{ and} \\ & m, n \geq 0, m+n \leq \frac{N}{2} \text{ and } m-n \leq \frac{N}{2} \\ 0, & \text{otherwise} \end{cases} \quad (3.39)$$

However, there are two major problems with this approach. One is the fact that there is no averaging being done (like straight periodogram) and therefore the variability will be high, especially since there are so many higher products begin computed.

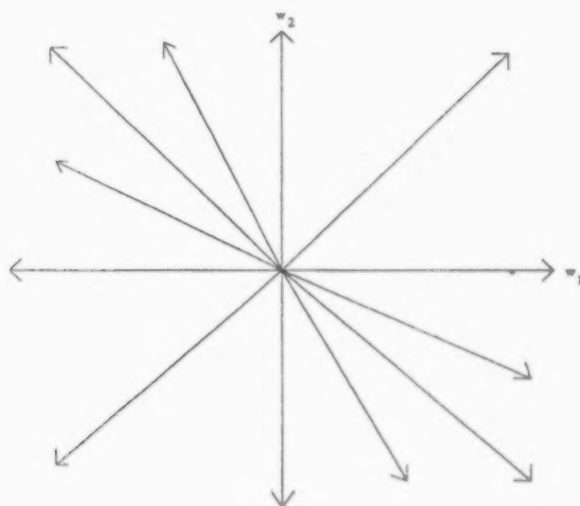


Figure 3.3: 12-slice bispectrum domain partition. Each slice completely describes the bispectrum

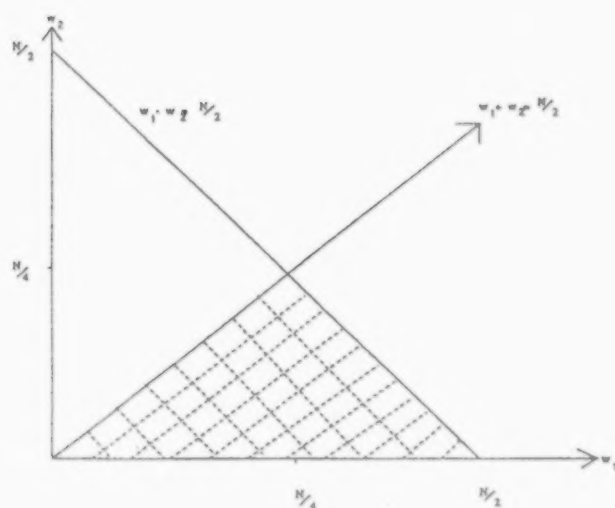


Figure 3.4: Zoom-in of first discrete bispectral slice.

Another much bigger problem, however, is that for large N , the number of points generated in the bispectrum domain is very very large ($\propto \frac{N^2}{16}$), and therefore using a single large FFT is very unrealistic. For example, 8192 input points will result in roughly 4.194 million raw bispectrum points to be processed by the classifier! Just generating this many points is too computationally demanding and for all intensive purposes, impossible. In addition, even if such large bispectra could be computed, due to the fact that there are products at differing frequencies, it is extremely difficult to selectively "discard" points to reduce input dimensionalities to within computable limits approximately under 700.

To get around these problems, a method suggested by Nikias can be used [15]. This method is the equivalent of the Welch approach for the periodogram, using smaller FFT sizes and taking averages. However, there is a loss of resolution but reduction of both variability and input dimensionality.

Consider again an N -point sequence, where $N = KM$, and both N and M are powers of two. As in the Welch periodogram procedure, the data can be segmented into K segments of M points and the Fourier transforms taken:

$$x^i(k) = x(k + iM), \quad 0 \leq k \leq M - 1, \quad 0 \leq i \leq K - 1 \quad (3.40)$$

and

$$Z^i(k) = \mathcal{F}\{x(k + iM)\}, \quad 0 \leq k \leq M - 1, \quad 0 \leq i \leq K - 1 \quad (3.41)$$

Now using the M Fourier transform sequences, an average discrete bispectrum can be computed as follows (Eq. 3.36):

$$\hat{B}(m, n) = \begin{cases} \frac{1}{K} \sum_{k=0}^{K-1} Z^k(m) Z^k(n) Z^{k*}(m+n), & \text{where } \omega_m = \frac{m}{2\pi M}, \omega_n = \frac{n}{2\pi M} \text{ and} \\ & m, n \geq 0, m+n \leq \frac{M}{2} \text{ and } m-n \leq \frac{M}{2} \\ 0, & \text{otherwise} \end{cases} \quad (3.42)$$

The estimate of the bispectrum can be further improved by weighting it with a window in the bispectral domain to reduce the bias. Gabr and Rao [17] state that an optimal

window (developed with minimum bias criteria) is given by:

$$W(m, n) = \begin{cases} \frac{\sqrt{3}}{\pi^2} \left\{ 1 - \frac{1}{\pi^2} (\omega_m^2 + \omega_n^2 + \omega_m \omega_n) \right\}, & \text{where } \omega_m = \frac{m}{2\pi M}, \omega_n = \frac{n}{2\pi M} \text{ and} \\ & m, n \geq 0, m + n \leq \frac{M}{2} \text{ and } m - n \leq \frac{M}{2} \\ 0, & \text{otherwise} \end{cases} \quad (3.43)$$

The above window can be applied after the averaging over the K estimates to get the final result. Nikias [15, p.878] states that for large N and M , the above windowed estimate is approximately unbiased:

$$E \{ \hat{B}(\omega_m, \omega_n) \} \cong B(\omega_m, \omega_n) \quad (3.44)$$

and the variance is asymptotically zero:

$$\sigma_{\hat{B}(\omega_m, \omega_n)}^2 \cong \frac{V}{K} P_{xx}(\omega_m) P_{xx}(\omega_n) P_{xx}(\omega_m + \omega_n) \quad (3.45)$$

where $P_{xx}(\omega)$ is the ideal power spectrum of the process and V is the window energy given by:

$$V = \sum_{m=-M}^{+M} \sum_{n=-M}^{+M} |W(m, n)|^2 \quad (3.46)$$

With this averaging/windowing technique, the number of points generated can be reduced drastically by choosing large K . However, this will give much lower resolution in the bispectral domain and hence may still pose problems for difficult classes which are close together. Apart from selectively averaging groups of points together, there is no way to further reduce input dimensionality to the classifier since the bispectra cannot be "constrained" to lie within a certain part of the bispectral slice. Hence the number of input features to the classifier using a bispectral preprocessor are considerably more than those using a periodogram-based preprocessor, and are given by:

$$\text{Number of features} = \sum_{i=0}^M \left(\frac{M}{2} - 2i \right) \quad (3.47)$$

This increase requires larger training and testing sets and also significantly more computational time and power.

It is postulated that the use of bispectra will extract more important discriminatory information for the non-Gaussian signals. The speech component in the analog signals is definitely non-Gaussian, and so is the random bitstream in the digital modulations. If it were certain that the signals are Gaussian distributed, then there would be no need for using this method since the bispectra of such signals is ideally zero [17].

Chapter 4

Pattern Classifiers

The pattern classifier takes an input feature vector, and given a database of labelled feature vectors (training set), computes its class. (A labelled feature vector is one whose correct class is specified. Feature vectors are generated by the preprocessor). There are many types of classifiers that can be used for this multi-class AMR problem. Standard conventional techniques include linear discriminant functions and k-nearest neighbour classifiers (k-NN). Recently, the growth in the neural networks field has led to the development of several neural network classifiers, such as backpropagation networks and radial basis function networks. It is decided to investigate the performance of the backpropagation neural network classifier in conjunction with the preprocessing paradigms chosen. This type of network is chosen because it is very versatile and is a standard against which to gauge further studies with different networks. The performance of the neural network is to be compared against that of a k-NN classifier, since, again, this is one of the more popular conventional classifiers. Detailed descriptions of both classifiers are given and implementation and optimization details discussed for the backpropagation network classifier.

4.1 K-Nearest Neighbour Classifier

If sufficient knowledge of the densities of the various classes is not known, then it is best to use a *non-parametric* technique. Such techniques assume no apriori knowledge

of the underlying distributions of the data and rely on a large training set. The k -NN approach is a such a method and basically exchanges the need to know the underlying distributions with that of knowing a large number of correctly classified feature vectors [18]. This method is a modification of the Parzen window estimate of a density function, which is more geared towards classification. In other words, rather than explicitly estimating the densities, it is only desired to know which one is larger than the others at a given point. The following describes this procedure:

Consider an n -dimensional input feature space, with N labelled training vectors, $\underline{x}_i$, $i = 0, \dots, N-1$. Given a vector, $\underline{x}$, its k -nearest neighbours are computed using a Euclidean distance metric:

$$d(\underline{x}, \underline{x}_i) = \sqrt{(x_1 - x_{i1})^2 + (x_2 - x_{i2})^2 + \dots + (x_n - x_{in})^2} \quad (4.1)$$

For an M -class problem, let $k_1, k_2, \dots, k_M$ be the number of nearest neighbours from each class, $\omega_1, \omega_2, \dots, \omega_M$ (ie. $k_1 + k_2 + \dots + k_M = k$). The decision rule is therefore simply [2, p.182]:

$$k_i = \max \{k_1, k_2, \dots, k_M\} \longrightarrow \underline{x} \in \omega_i \quad (4.2)$$

This rule is valid since the Parzen window method estimates the class-conditional density as:

$$\hat{p}_i(\underline{x}) = \frac{k_i}{N_i} \frac{1}{A(k, N, \underline{x})} \quad (4.3)$$

where N_i is the number of training samples in the i -th class and $A(k, N, \underline{x})$ is the volume of a hypersphere of radius r , independant of the class [10]. So basically, the k -NN approach computes the k -nearest neighbours of an inputted test vector and then assigns the vector to the class corresponding to the most prevalent neighbour. Ties are resolved by using uniform distribution to judge between the conflicting classes. Generally, if k is chosen to be odd, ties do not occur.

Assuming large N , it can be said that a point $\underline{x}$ and its nearest neighbour, $\underline{x}_k$ are very close, yielding the following approximation:

$$P(\omega_i | \underline{x}_k) \approx P(\omega_i | \underline{x}) \quad (4.4)$$

Using this limiting case, Fukunaga [2, p.182] states that the probability of misclassification using the 1-NN for $\underline{x}$ is given by the sum of the conditional errors.

$$\epsilon(\underline{x}) = P(\omega_1 | \underline{x}) \sum_{j=1, j \neq 1}^M P(\omega_j | \underline{x}) + \dots + P(\omega_M | \underline{x}) \sum_{j=1, j \neq M}^M P(\omega_j | \underline{x}) \quad (4.5)$$

Using the fact that $P(\omega_1 | \underline{x}) + P(\omega_2 | \underline{x}) + \dots + P(\omega_M | \underline{x}) = 1$, Eq. 4.5 can be written as:

$$\epsilon(\underline{x}) = 1 - \sum_{j=1}^M P^2(\omega_j | \underline{x}) \quad (4.6)$$

Note that the Bayes conditional error for $\underline{x}$ is given by:

$$\epsilon^*(\underline{x}) = 1 - \max_j \{P(\omega_j | \underline{x})\} = 1 - P(\omega_l | \underline{x}) \quad (4.7)$$

where $P(\omega_l | \underline{x}) = \max_j \{P(\omega_j | \underline{x})\}$. Combining this with the Schwarz inequality it can be derived:

$$(M-1) \sum_{j=1, j \neq l}^M P^2(\omega_j | \underline{x}) \geq \left\{ \sum_{j=1, j \neq l}^M P(\omega_j | \underline{x}) \right\}^2 = \{1 - P(\omega_l | \underline{x})\}^2 = \epsilon^{*2}(\underline{x}) \quad (4.8)$$

Now by adding the quantity $(M-1)P^2(\omega_l | \underline{x})$ to each side of Eq. 4.8,

$$(M-1) \sum_{j=1}^M P^2(\omega_j | \underline{x}) \geq \epsilon^{*2}(\underline{x}) + (M-1)(1 - \epsilon(\underline{x}))^2 \quad (4.9)$$

Rearranging 4.9, the error can be bounded from above by

$$\epsilon(\underline{x}) \leq 2\epsilon^*(\underline{x}) - \frac{M}{M-1}\epsilon^{*2}(\underline{x}) \quad (4.10)$$

To compute a bound on the overall error rate, the mean of Eq. 4.10 is taken giving

$$\epsilon \leq 2\epsilon^* - \frac{M}{M-1}\epsilon^{*2} \quad (4.11)$$

where ϵ^* is the true Bayes error rate, and ϵ is the ideal error rate of the k-NN. This is a very important result, stating that given a large training set size, N , the error rate given by the k -NN approach is always less than twice the Bayes error rate. The larger the training set is, the better the performance of the k -NN. However, classification time increases linearly with N , and for large datasets of high dimensionality, can be very large. Extremely indepth analysis shows that the error of the k -NN for a "large" dataset improves with increasing k . Intuitively this can be explained by the fact that more samples are being used to estimate the conditional probabilities. Since very dense sampling is assumed, these estimates are more accurate.

4.2 Backpropagation Neural Network Classifier

With the increase in modern computing power, there has been a rapid expansion in the development and application of neural networks. Specifically, the backpropagation network has become the workhorse of many neural network-based products and many variants of it are now in use. This type of network can also be used for classification problems given that there is enough computing power and training data.

Basically, a backpropagation network is just an interconnection of layers of computing units (neurons) with weights. A computing unit consists of a summer which adds the weighted inputs, an offset bias, followed by a non-linear smoothing function (see Figure 4.2). The number of layers (excluding input and output) in the network is arbitrary, with two typically being sufficient for most problems. If each unit in a given layer is connected to all the units in the layers before and after it, the network is fully-connected. Many times, not all weights are required and therefore methods have to be devised to selectively remove some of the connections. Apriori knowledge and designer intuition are some of the more adhoc methods of doing such "pruning". A more refined technique based on a cost criteria is presented later on. Figure 4.1 depicts a sample two-hidden layer network. The input to the net is a feature vector, extracted by the preprocessor, and the output is an approximation to the desired output (class). Therefore, it can be said that the network performs a non-linear mapping from the input space to the output space.

A major part of using the backpropagation network is the training phase, in which large amounts of labelled training vectors are used to train the network. Training essentially involves iteratively adapting the weights of the network using an error backpropagation technique. This technique presents an input vector (or groups of input vectors) to the network, computes the output(s), and then compares the output(s) to the desired responses, thereby generating an error signal. This error signal is then propagated backwards through the network to recursively update the weights, and hence the name. The technique aims to minimize the squared error between the output of the network and the desired output. A quick review of the backpropagation equations is now given. For detailed derivations, the interested reader is referred to

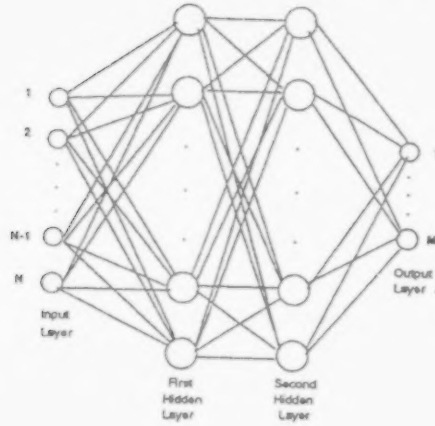


Figure 4.1: Fully-connected two-hidden layer backpropagation network

the references [19], [25]. Issues regarding training, testing, and implementation are also addressed.

4.2.1 Summary of Backpropagation Algorithm

Consider an N -dimensional input vector, presented at iteration n , $\underline{x}(n)$, which generates an M -dimensional output vector, $\underline{y}(n)$. Defining the ideal output for $\underline{x}(n)$ as $\underline{d}(n)$, an error vector can be defined as:

$$\underline{\varepsilon}(n) = \underline{d}(n) - \underline{y}(n) \quad (4.12)$$

The error is then defined from $\underline{\varepsilon}(n)$ as:

$$E(n) = \frac{1}{2} \|\underline{\varepsilon}(n)\|^2 = \frac{1}{2} \sum_{j=1}^M (d_j(n) - y_j(n))^2 \quad (4.13)$$

The output of the network, $y_j(n)$, is the smoothed sum of weighted inputs, to the j -th node, $v_j(n)$. The smoothing function used is the sigmoid nonlinearity:

$$y_j(n) = \phi(v_j(n)) = \frac{1}{1 + e^{-v_j(n)}} \quad (4.14)$$

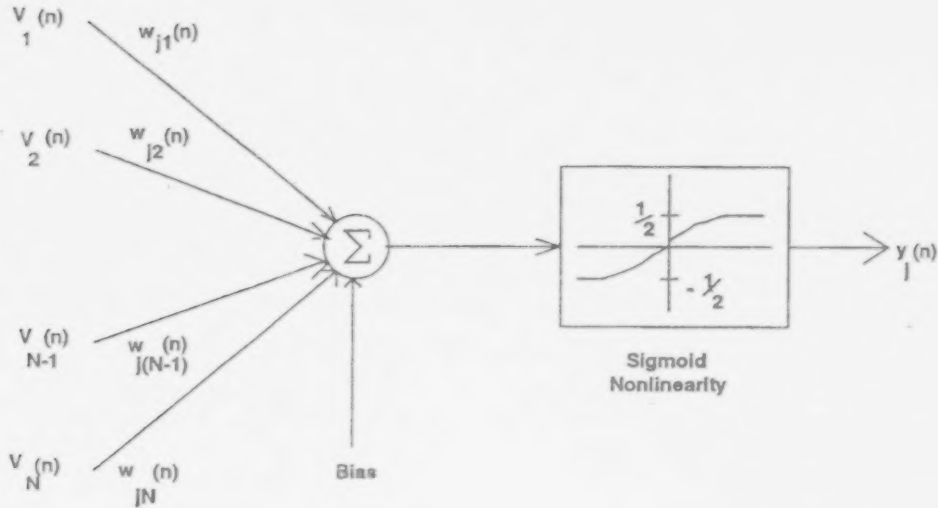


Figure 4.2: A computing element, neuron, of a backpropagation network

and

$$v_j(n) = \sum_i w_{ji}(n)x_i(n) - \theta_j \quad (4.15)$$

where $x_i(n)$ are the outputs of the last hidden layer, $w_{ji}(n)$ is the weight connecting the j -th output neuron to the i -th neuron in the last hidden layer, and θ_j is an offset bias.

To update the weights, the gradient of the error surface with respect to the weights can be taken and its negative used to modify the weights accordingly. Note that the bias is also considered as a weight, one which has unity input. Applying the chain rule to differentiate Eq. 4.13 with respect to the weights, one gets:

$$\frac{\delta E(n)}{\delta w_{ji}(n)} = -\delta_j(n)y_i(n) \quad (4.16)$$

where $\delta_j(n)$ is a modified error signal for the j -th output neuron:

$$\delta_j(n) = \phi'_j(v_j(n))\epsilon_j(n) = y_j(n)(1 - y_j(n))\epsilon_j(n) \quad (4.17)$$

It can be shown that for *non-output* neurons (ie. hidden layer neurons), the derivative

is still given by Eq. 4.16 except that now, $\delta_j(n)$ is defined as:

$$\delta_j(n) = \phi'_j(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \quad (4.18)$$

Here, $v_j(n)$ is the weighted sum outputs from the the hidden layer previous to the one being considered, and $w_{jk}(n)$ is the value of the weight connecting the j -th neuron in the hidden layer to the k -th neuron in the previous hidden layer. Using the delta-rule to update the weights, the weight update is:

$$\Delta w_{ji}(n) = -\eta \frac{\delta E(n)}{\delta w_{ji}(n)} \quad (4.19)$$

yielding

$$w_{ji}(n+1) = w_{ji}(n) - \eta \frac{\delta E(n)}{\delta w_{ji}(n)} = w_{ji}(n) + \eta \delta_j(n) y_i(n) \quad (4.20)$$

Here, η is the learning rate, which controls the step size of the weight updates. Typically, small values of η should be used to allow good descent into minima. Using large learning rates will result in an unstable network with the weights "jumping" around and not settling to a good enough minima. Note however, that a small learning rate will slow down the convergence of the algorithm, and hence may not be very feasible. A simple technique which can be used to speed up the rate of convergence is the addition of a momentum term in Eq. 4.20:

$$w_{ji}(n+1) = w_{ji}(n) + \eta \delta_j(n) y_i(n) + \alpha \Delta w_{ji}(n-1) = -\eta \sum_{k=0}^n \alpha^k \frac{\delta E(n-k)}{\delta w_{ji}(n-k)} \quad (4.21)$$

where α is the momentum factor, between 0 and 1. This new technique basically takes in account past weight changes by using an exponential weighting factor. When successive derivatives have the same sign, the weight change is increased, and when they have differing signs, it is decreased. This enables the algorithm move with "larger" steps on smooth error surfaces and smaller ones on rough surfaces to prevent overshooting possible minima.

Despite the addition of the momentum factor, the algorithm can still be very slow, especially for large networks. Since weight changes are generally small per pattern presented, as a modification, one can accumulate the derivatives over a group of training patterns and then update the weights rather than doing so after each

pattern. During this accumulation, the same weights from the previous update are used. This method is called the *batch* update method, and the group size is called the *batch size*, P . (Note that for $P = 1$, the method is equivalent to using Eq. 4.21, called online update). The equations for this new approach can be easily derived from those for the regular backpropagation algorithm. With a slight change of notation, $y_j^p(n)$ is defined as the output of the j -th neuron for the p -th input vector, $\underline{x}^p(n)$. $\underline{d}^p(n)$ is the desired output for $\underline{x}^p(n)$, and the error vector is given by:

$$\underline{\varepsilon}^p(n) = \underline{d}^p(n) - \underline{y}^p(n) \quad (4.22)$$

The weighted sum of inputs to the j -th neuron for the p -th pattern is then given by:

$$v_j^p(n) = \sum_i w_{ji}(n) x_i^p(n) - \theta_j \quad (4.23)$$

With the batch method, the error cost function, $E(n)$, basically becomes Eq. 4.13 summed over the P patterns:

$$E(n) = \frac{1}{2} \sum_{p=1}^P \|\underline{\varepsilon}^p(n)\|^2 = \frac{1}{2} \sum_{p=1}^P \sum_{j=1}^M (d_j^p(n) - y_j^p(n))^2 \quad (4.24)$$

The weight update for this method is given by:

$$\Delta w_{ji}(n) = -\eta \frac{\delta E(n)}{\delta w_{ji}(n)} \quad (4.25)$$

where the derivatives are now summed

$$\frac{\delta E(n)}{\delta w_{ji}(n)} = \sum_{p=1}^P -\delta_j^p(n) y_i^p(n) \quad (4.26)$$

Here the backpropagated error quantities, $\delta_j^p(n)$, are given by:

$$\delta_j^p(n) = \begin{cases} \phi_j'(v_j^p(n)) \varepsilon_j^p(n), & \text{if neuron } j \text{ is output} \\ \phi_j'(v_j^p(n)) \sum_k \delta_k^p(n) w_{kj}(n), & \text{otherwise} \end{cases} \quad (4.27)$$

In summary, the training procedure for the batch algorithm is given as follows:

1. Initialize the Weights

Using a random uniform distribution, ranging from $-\kappa$ to $+\kappa$, choose random values for all the weights. Set learning rate to small value ($\eta \leq 0.1$) and momentum to high ($\alpha \geq 0.7$).

2. Forward Pass

Present P input training vectors, $\underline{x}^p(n)$, $p = 1, \dots, P$, and compute their outputs $\underline{y}^p(n)$. Compute error vectors using Eq. 4.22.

3. Backward Pass

Propagate error signals backwards and compute weight updates using Eqs. 4.21 and 4.26 for all weights. Updates are done layer by layer until one reaches the weights connecting the input and first hidden layers.

4. Repeat Step 2 until change in error sufficiently small

As mentioned previously, the benefit of this approach is the speedup in computation time, since the weight updates are done less frequently. This technique also gives a better estimate of the true estimate of the gradient by summing over a group of patterns. However, since the the gradient estimate is more "smoothed", the chances of the algorithm getting stuck in a local minima are also increased. Neither the batch nor online update algorithm is guaranteed to converge to a global minima in the weight-space. It is hoped that the algorithm does, however, find a low enough minima to give satisfactory results for the problem at hand.

4.2.2 Using Backpropagation Networks

Even though the algorithms are well-known, there are many heuristics and other rule-of-thumb measures involved in successfully applying backpropagation networks to a given problem. Some of these issues are now addressed.

Backpropagation networks can be applied to many sorts of problems: in our case, the problem is classification of inputs into one of multiple classes. Here the inputs are merely the outputs of the preprocessor, but how does one specify the desired outputs for training? For example, should there be one output neuron per class, or one output neuron for all classes? There is sufficient theory [20], [21] indicating that if neurons are assigned for *each* class, then their outputs represent estimates of the

aposteriori probabilities of the inputs belonging to that particular class. In other words, consider a "properly" trained network with input vector $\underline{x}$, and M outputs (ie. output vector $\underline{y}$ is M -dimensional). Then the components of the output vector approximate the following:

$$y_i \simeq P(\omega_i | \underline{x}) \quad (4.28)$$

where there are M classes, ω_i . Note that for Eq. 4.28 to hold, it is assumed that the desired outputs are specified in a 1-0 manner:

$$\underline{d}_n = \underline{c}_i, \text{ if } \underline{x}_n \in \omega_i \quad (4.29)$$

where $\underline{c}_i$ is the i -th column of the $M \times M$ identity matrix, $\underline{I}_M$. This therefore gives a simple decision rule for classifying test patterns: assign the pattern to the class corresponding to the neuron with maximum output. Since the range of the sigmoid non-linearity is from 0 to 1, specifying the desired outputs in the above manner is valid. Ideally, the output for the correct neuron should be 1 and 0 for all other neurons. However, since machine precision will never reach these ideal bounds, the weights will be driven into saturation. A heuristic states that the desired outputs should be specified as $1 - \delta$ for 1 and δ for 0. δ is a small positive constant which should be well within the range of machine precision to prevent the algorithm from driving the weights into saturation (ie. summation of weighted inputs to $\pm\infty$). In addition, it is better to scale the input vectors to lie within a certain magnitude to prevent overflows/underflows in the arithmetic. This is very critical since the WARP machine being used is only single precision.

Another issue involves the choosing the learning rates and momentum values. Generally, large momentum values should be used to accelerate the movement in smooth error surfaces and also to avoid letting the most recent gradient dominate the weight change. For learning rates, as mentioned before, the smaller the values, the less likely the algorithm is to skip potential minima. However, it is even better to specify different learning rates for the various weights in the network. A good rule of thumb is to specify the learning rate as being inversely proportional to the fan-in of the neurons of the layer. In other words, for a fully connected network, the learning rate for the input-hidden layer weights will be inversely proportional to the number

of input neurons, and that for the hidden-output weights, inversely proportional to the number of neurons in the last hidden layer. This helps keep the outputs of the summers in the neurons at the same respective levels and also helps avoid saturation in neuron outputs. As an oversimplified example, consider a network with just 2 inputs, 4 hidden neurons, and 2 output neurons. The fan-in for the hidden layer nodes is 3 (2 inputs plus bias) and similarly the fan-in for the output nodes is 5. The ratio of the hidden to output node fan-in is then 0.6. This implies that the learning rate for the hidden-output layer weights should be 0.6 times that for the input-hidden weights.

A major issue concerning the application of backpropagation networks is the size of the training set. Large training sets are required to ensure that enough discriminatory information is available for the network to perform properly. It is very hard to give a closed form figure on this set size, although some loose bounds have been developed. Nevertheless, in most cases even these figures are hard to meet practically. Tradeoffs have to be made in reality to train networks with smaller datasets.

When using neural networks, the hope is that the network correctly classify a large portion of input pairs never used in training. This ability is called *generalization*. Generalization can be measured by testing the network on a large test set, completely independent of the training set. It is obvious that the number of training examples has to be at least greater than the number of parameters (weights) in the network. Basically, the network is just learning a mapping from the input space, $\mathbb{R}^N$, to the output space, $\mathbb{R}^M$. This can be likened to a high-order polynomial curve fitting problem, in which the number of data points given has to be at least as large as the order of the fit. Baum and Haussler [23] state that for the error to be less than $\frac{\epsilon}{2}$ on a training set, m training examples have to be used:

$$m \geq \frac{32W}{\epsilon} \ln \left(\frac{32n}{\epsilon} \right) \quad (4.30)$$

where W is the number of weights and n is the number of neurons in the network. This implies [19] that in general, the training set size is directly proportional to the number of weights in the network and inversely proportional to ϵ , ie.

$$m \geq \frac{W}{\epsilon} \quad (4.31)$$

Since these bounds are very general and assume no apriori information about the training data, strictly abiding by them for large W (order of 10^3 or more) and even modest error rates will give impractical training set sizes. Such large sizes will easily overshoot the available memory resources on the computing systems to be used. It could very well be that much fewer training examples are required to give satisfactory results, depending on the nature of the problem (ie. feature set, and separation of classes). For example, Ahmad and Tesauro [24] experimentally show that the error rate on a network trained for the majority function levels off after a certain training size. Beyond this "threshold", additional training vectors give little reduction in the error rate. This method suggests the use of an experimental approach, in which the training set size is increased iteratively until the error rate "levels off" to satisfactory value. To get a good measure of generalization, the error rate should be measured on a very large test set, completely independent of that used for training. The issue of measuring classifier performance is discussed in Section 4.3.

It is mentioned that the test set should be completely independent of the training set to get a true measure of performance. However, at the same time, a set (validation) is required during training to periodically check the performance of the network. In this method, at fixed intervals during training, the network is periodically tested on the validation set to see whether or not training should be continued. It is too simple to just consider the slope of the error vs. iteration plot and stop when the slope falls below a chosen threshold, close to zero. There is no guarantee that a "flat" error plot means that the network is properly trained. Initially, as training starts, the performance on the validation set improves. However, if training is carried on too long, the network begins to "memorize" the training patterns (ie. interpolation), and the generalization decreases (ie. increase in error on independent test set). This phenomena is called overlearning and therefore it is important to stop the training before it occurs (see Figure 4.3). In this scenario, the validation set is not entirely independent of the training, since it is used to provide feedback on when to stop training. Therefore there should be another test set, independent of both training and validation sets, which will finally be used to get a true measure of the network's performance. This set is used only after the network training is fully complete. It

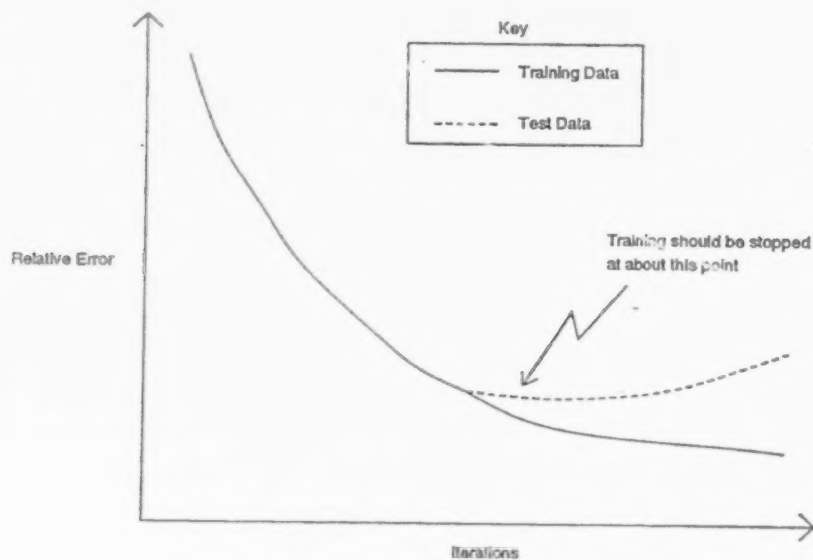


Figure 4.3: Overtraining in backpropagation networks

is assumed that all sets (training, validation, test) represent each class equally. This prevents any biasing in the training or measurement of performance towards specific classes. On an implementation note, it is best to present the training vectors from the differing classes in a random order.

4.2.3 Network Pruning: Optimal Brain Damage

"Most successful applications of neural networks to real-world problems have resulted in using highly structured networks of rather large size." [26] In many cases, the large number of parameters (weights) being used may very well result in overfitting of the training data: in other words, the networks begin fitting the noise in the data. This will obviously lead to poor generalization and hence reduced effectiveness. As a result, techniques to reduce the number of parameters are required. In addition, a reduced parameter count will mean more effective use of the training data, since the same

amount of vectors are being used to train fewer weights. No doubt that implementing smaller parameter networks is also easier. Various weight reduction/elimination techniques exist for networks, but a lot of them are adhoc. It is desired to have a more rigorous method, even at the expense of increased computation requirements.

Le Cun et al. [26] have devised a network pruning technique called optimal brain damage (OBD), based on a second-derivative cost heuristic. The idea behind this procedure is to take a fully trained network and delete a substantial number of weights from it without undermining and possibly improving generalization. "OBD aims to use second-order derivatives of the error function, $E(n)$ (Eq. 4.13), to make a tradeoff between network complexity (number of weights) and training error performance." [19] The development of the algorithm hinges around the expansion of the error function in a Taylor series. A summary of the OBD algorithm is now given.

To properly describe OBD, a Taylor series expansion of $E(n)$ is required. Haykin [19] defines $\delta u_k(n)$ as the perturbation in the value of parameter $u_k(n)$. Generally, $u_k(n)$ is a weight in the network, ie. $u_k(n) = w_{ji}(n)$ for all $(j, i) \in V_k$, V_k being a complete set of index pairs. Using this notation, the Taylor series expansion of $E(n)$ is given by:

$$\begin{aligned} \delta E(n) = & \sum_k \frac{\delta E(n)}{\delta u_k(n)} u_k(n) + \frac{1}{2} \sum_k h_{kk}(n) \delta u_k^2(n) + \\ & \frac{1}{2} \sum_j \sum_{k_j \neq k} h_{jk}(n) \delta u_j(n) \delta u_k(n) + O(\|\delta u_k(n)\|^3) \dots \end{aligned} \quad (4.32)$$

where $h_{jk}(n)$ is the the jk -th element of the Hessian matrix of $E(n)$:

$$h_{jk}(n) = \frac{\delta^2 E(n)}{\delta u_j(n) \delta u_k(n)} \quad (4.33)$$

It is desired to find the subset of parameters whose deletion will cause the least increase in the value of $E(n)$ (ie. minimize $\delta E(n)$). Typically, for networks containing thousands of weights, it is impossible to solve the above problem. The Hessian matrix itself will be too large to handle on most computers. As a result, some assumptions are made to make the problem easier to solve [19]:

1. Weights are deleted only after training

This means that the weights already reside at a local/global minima on the $E(n)$ surface and therefore the first order derivatives in Eq. 4.32 can be neglected.

2. Hessian matrix is diagonally dominant

This allows one to discard off diagonal elements of the Hessian matrix contributing to the Taylor series. In otherwords, the double summation in Eq. 4.32 can also be discarded.

3. Quadratic Approximation

Assuming that $E(n)$ is quadratic around a local/global minima, the higher order terms of Eq. 4.32 can be neglected.

With these assumptions, the series expansion reduces to:

$$\delta E(n) = \frac{1}{2} \sum_k h_{kk}(n) \delta u_k^2(n) \quad (4.34)$$

Le Cun et al. [26] define a "saliency" measure for each weight based on Eq. 4.34 as:

$$s_{u_k}(n) = \frac{1}{2} \sum_k h_{kk} u_k^2(n) \quad (4.35)$$

The saliency is therefore used as a deletion measure. Weights whose saliencies are below a certain threshold are either decayed or deleted (ie. zeroed). The network is then retrained after these adjustments and then the procedure repeated iteratively until a satisfactory performance level (ie. percentage deletion vs. generalization) is attained.

The second derivative ($h_{kk}(n)$ in Eq. 4.34) can easily be derived from the equations for the regular backpropagation algorithm. Consider the online update case (ie. batch size $P = 1$, error criteria given by Eq. 4.13), where the first order derivatives are given by Eq. 4.17 for the last layer of weights and by Eq. 4.18 for all other layers. For the last layer weights (ie. hidden-output), Eq. 4.17 can be differentiated again to get a second derivative:

$$\begin{aligned} \Gamma_{w_{ji}(n)}^{\text{output}} &= \frac{\delta^2 E(n)}{\delta w_{ji}^2(n)} \\ &= -2y_i(n) \frac{\delta}{\delta w_{ji}(n)} (\phi'(v_j(n)) e_j(n)) \end{aligned}$$

$$= -2y_i(n) \left(e_j(n) \frac{\delta \phi'(v_j(n))}{\delta w_{ji}(n)} + \phi'(n) \frac{\delta e_j(n)}{\delta w_{ji}(n)} \right) \quad (4.36)$$

where $v_j(n)$, $y_j(n)$, and $e_j(n)$ are identical to the quantities defined in the regular backpropagation algorithm. It can be easily shown [19] that the above expression is equivalent to:

$$\frac{\delta^2 E(n)}{\delta w_{ji}^2(n)} = 2 \left(\phi'(v_j(n)) \right)^2 y_i^2(n) - 2\phi''(v_j(n)) e_j(n) y_i^2(n) \quad (4.37)$$

where

$$\begin{aligned} \phi''(v_j(n)) &= \frac{\delta^2(v_j(n))}{\delta v_j^2(n)} \\ &= y_j(n) (1 - y_j(n)) (1 - 2y_j(n)) \end{aligned} \quad (4.38)$$

Similarly, the second derivative for hidden layer weights can be computed by differentiating Eq 4.18 again. The resulting expression for the second derivative is:

$$\begin{aligned} \Gamma_{w_{ji}(n)}^{hidden} &= \frac{\delta^2 E(n)}{\delta w_{ji}^2(n)} \\ &= -2 \left(\phi'(v_j(n)) \right)^2 y_i^2(n) \sum_k w_{kj}^2(n) \varphi_k(n) \phi''(v_j(n)) \\ &\quad - 2\phi''(v_j(n)) \varphi_j(n) y_i^2(n) \end{aligned} \quad (4.39)$$

where $\varphi_j(n)$ is given by

$$\varphi_j(n) = \sum_k \delta_k(n) w_{kj}(n) \quad (4.40)$$

and $\delta_k(n)$ are the modified error terms from the regular backpropagation algorithm. A lot of tedious details in deriving Eq. 4.39 have been omitted, and the interested reader is referred to the references [19], [26] for detailed derivations.

As is the case with the first order derivatives, if second order derivatives are summed over a batch, smoother estimates are obtained. This will also speed up the algorithm tremendously since computing second order derivatives is considerably more computationally intensive than computing first order derivatives. The equations governing the second derivatives are easy to extend for the batch case and the results are (for batch size P):

$$\frac{\delta^2 E(n)}{\delta w_{ji}^2(n)} = \begin{cases} \sum_{p=1}^P \Gamma_{w_{ji}(n)}^{p,output}, & \text{if neuron } j \text{ is output} \\ \sum_{p=1}^P \Gamma_{w_{ji}(n)}^{p,hidden}, & \text{otherwise} \end{cases} \quad (4.41)$$

Note that $\Gamma_{w_{ji}(n)}^{p_{\text{output}}}$ and $\Gamma_{w_{ji}(n)}^{p_{\text{hidden}}}$ are the double derivatives (Eq. 4.36, Eq. 4.39) for the p -th batch vector. It should be noted that for batch mode, the $\delta E(n)$ is now the perturbation in the the summed error, Eq. 4.24.

In summary, the OBD algorithm is given by the following major steps:

1. Train network

Using the batch backpropagation algorithm, train to obtain suitable performance.

2. Compute second derivatives of error

Evaluate Eq. 4.41 for all weights, making sure that batch size, P , is the same as that used in the batch backpropagation (Step 1).

3. Compute saliences

Using results of above step, evaluate Eq. 4.35.

4. Delete low-saliency weights

All weights whose saliences are below a certain threshold value are set to zero.

5. Goto Step 1

Repeat procedure until either satisfactory error or deletion ratio is reached.

It should be noted that Step 4 in the above procedure can be time-consuming since it involves experimentation to determine good threshold values. Choosing correct threshold values is very important since large thresholds will give catastrophic results, deleting too much weights to be useful. Also, note that in Step 4 weights whose saliences are below the threshold are zeroed and not decayed. This is done because the decaying may take many iterations to reduce redundant weights to negligible values. Since double derivatives are being computed on each iteration, computation time increases drastically and it is therefore not feasible to decay weights.

4.3 Measuring Classifier Error Rates

Typically, the ideal error rate for a classifier, ϵ , cannot be determined. Rather, an estimate of this error rate, $\hat{\epsilon}$, is made using a simple error counting procedure on a large, independant test set. It can therefore be said that $\hat{\epsilon}$ itself is a random variable.

Fukunaga [2] shows that by using the properties of a binomial distribution, the mean and variance of this variable are given by:

$$E\{\hat{\epsilon}\} = \epsilon \quad (4.42)$$

and

$$\sigma_{\hat{\epsilon}}^2 = \sum_{i=1}^M P(\omega_i) \frac{\epsilon_i(1 - \epsilon_i)}{N_t} \quad (4.43)$$

where N_t is the number of *test* feature vectors and ϵ_i is the conditional error rate for class ω_i for the k-NN:

$$\epsilon = \sum_{i=1}^M P(\omega_i) \epsilon_i \quad (4.44)$$

Assuming equal apriori class probabilities, the above expression simplifies to:

$$\sigma_{\hat{\epsilon}}^2 = \sum_{i=1}^M \frac{\epsilon_i(1 - \epsilon_i)}{MN_t} \quad (4.45)$$

Rather than evaluating the above expression, a faster variance figure can be computed at the expense of a looser bound (ie. higher variance). It can be shown [2, p.148] that Eq. 4.45 is a tighter bound than:

$$\sigma_{\hat{\epsilon}}^2 = \frac{\epsilon(1 - \epsilon)}{N_t} \quad (4.46)$$

Using $\hat{\epsilon}$ to estimate $\sigma_{\hat{\epsilon}}^2$, confidence intervals for the estimate can be computed. Specifically, assuming large N_t and that $\hat{\epsilon}$ is Gaussian distributed, the true error is bounded by the estimated quantities in the following manner [7], [8]:

$$\hat{\epsilon} - z\sigma_{\hat{\epsilon}} \leq \epsilon \leq \hat{\epsilon} + z\sigma_{\hat{\epsilon}} \quad (4.47)$$

where z is determined by a Gaussian distribution and is the required confidence level. Using the estimate of the variance, this confidence is given by roughly

$$\hat{\epsilon} - z\sqrt{\frac{\hat{\epsilon}(1 - \hat{\epsilon})}{N_t}} \leq \epsilon \leq \hat{\epsilon} + z\sqrt{\frac{\hat{\epsilon}(1 - \hat{\epsilon})}{N_t}} \quad (4.48)$$

The above analysis therefore is a good practical method of obtaining confidence intervals on the performance of the classifier. It should be noted that this method is

very general and is independent of the classifier used. It is obvious that the more test vectors that are available (ie. larger N_t), the more accurate these the error rate and variance estimates will be. However, computation time increases linearly with N_t , and for high dimensionalities and big datasets, can be very large.

Chapter 5

Implementation of Backpropagation Networks

The backpropagation network is implemented on the WARP systolic array processor. The WARP is a high performance computer developed at Carnegie Mellon University and built by General Electric. The speed of the WARP greatly reduces the time involved in training a network, and can reach almost real-time speed for classification of preprocessed vectors (forward pass). A brief description of the WARP is given, and then the implementations of the both the regular batch backpropagation algorithm and the optimal brain damage algorithms discussed.

5.1 Overview of the WARP Systolic Array

Figure 5.1 shows an overview of the WARP. The machine consists of a vector of 10 computing cells, with each cell being connected to its neighbour via two data channels, X and Y. Each cell is capable of 10 MFLOPS (million floating point operations per second) peak performance and the channels can handle rates of upto 80 Mbytes/second. Note that only the Y channel is bidirectional. The WARP has its own specialized programming language called W2. The W2 compiler generates optimized object code for each cell and for the interface system. The syntax of this

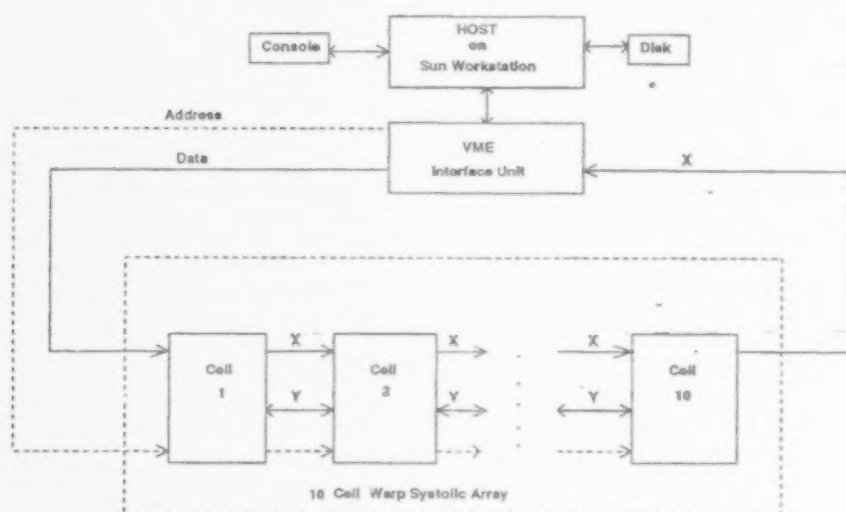


Figure 5.1: Overview of WARP Systolic Array Processor Setup

language is somewhat similar to PASCAL except for the fact that a lot of the command set is not present. In essence, W2 is very simple, consisting of a rudimentary command set along with some functions added to facilitate inter-cell communication. The mathematical functions available are extremely limited, being restricted to simply add/subtract and multiply/divide. As a result, the sigmoid has to be computed by using interpolation tables loaded in the cell memories. The interface unit, which comprises of a set of buffers, sends and retrieves data from the WARP. This unit connects to the *SUN Unix* system and allows users to access the WARP through a set of specialized WARP function libraries (WARP monitor). Hence, by using these libraries, users can write code in *C* to pass data to the interface unit, run the WARP, and retrieve the results. Detailed descriptions of the WARP can be found in the references ([27], [28], [29]). Generic details about the implementation of the backpropagation algorithm and the optimal brain damage algorithm on the WARP are now presented.

5.2 The Batch Backpropagation Algorithm

In order to get sufficient speed in the implementation, even on a parallel machine, a significant batch size has to be chosen. Small batch sizes will result in the inter-cell data channels getting too congested with traffic and hence slowing down the algorithm. This makes the online update method very slow and infeasible. The backpropagation network software developed at Carnegie Mellon University has a fixed batch size of 90 patterns (ie. $P = 90$ in Eq. 4.26). This size has been chosen by carefully experimenting with varying batch sizes, keeping the data channels as busy as possible, but ensuring that they are not getting overloaded. Hence data partitioning is done on the input data into groups of 90 vectors, with the WARP further partitioning the data among the first 9 cells, each cell receiving 10 distinct patterns. The first 9 cells basically perform the forward and backward passes of the algorithm for their 10 input patterns, and the 10th cell acts as an overall controller, performing the actual weight updates and transfer to the host.

Figure 5.2 shows a detailed description of the backpropagation algorithm on the WARP for each iteration. At the start of every iteration, the weights and interpolation tables are piped into all the cells of the WARP from the host program. This is done by passing the values to cell 1, which stores copies for itself and passes on copies (X channel) to cell 2. This is done till all cells are loaded. Next, a batch of 90 training vectors is taken from the training set and sent to cell 1. This cell will take 10 vectors and pass the remaining ones on to cell 2. This procedure is repeated until the all first 9 cells have 10 training vectors. Once all the cells have their training vectors, the forward pass is computed. All cells run independently of each other until the forward pass is complete. At the end of the forward pass, each cell (1-9) waits for its left-neighbour to pass on an error magnitude. When it receives this value, it adds its own error (pertaining to the 10 training vectors it has), and passes it on. This waiting for incoming error values acts as a synchronizing mechanism between adjacent cells, ensuring that no one cell runs without some sort of control from its neighbours. The error is eventually received at the 10th cell which sends it to the host program for a screen display. The last major step for each cell involves recursively going back

through the layers and computing all the derivatives for its 10 patterns.

To speed up the backpropagation algorithm, the $\delta_j^p(n)$ values, Eq. 4.27, computed for a given layer can be stored and reused for computing weight changes in previous layers. Specifically, first the derivatives pertaining to the hidden-output weights are computed (Eq. 4.16). The $\delta_j^p(n)$ computed for this layer (Eq. 4.17) can be stored and used to compute derivatives for weights for the next embedded layer (input-hidden or hidden-hidden), Eq. 4.18. In addition, quantities computed in the forward pass ($v_j^p(n)$, $y_j^p(n)$, $e_j^p(n)$) are stored for use in the backpropagation. This storage of intermediate quantities greatly reduces computation time since these values are not needlessly recomputed repeatedly for each weight. Then, as is done for the error, these updates are summed and passed to the last cell. Using the momentum weight update, the 10th cell updates the weights and passes them back to the host program. On the next iteration, these new weights along with another batch of 90 patterns are sent to cell 1 and the procedure repeated. The host program running in *C* has an interactive shell which displays the error per iteration and allows the user full flexibility in adjusting learning rates, momentum values and controlling training. For testing vectors, a similar program for the WARP also exists. The algorithm for this program is very similar to that depicted in Figure 5.2 except for the fact that only the forward pass is computed and the results passed on to the last cell.

5.3 The Batch Optimal Brain Damage Algorithm

The batch optimal brain damage on the WARP array processor has a very similar implementation to that of the batch backpropagation, with changes being made to the W2 code to compute the second derivatives (ie. Step 5, cells 1-9, Figure 5.2). As in the batch backpropagation case, at the start of each iteration the weights and sigmoid interpolation tables are piped in and each cell takes 10 incoming training vectors. The first order derivatives are computed in exactly the same manner as the regular implementation. Again, as is done in the regular implementation, certain quantities computed for the first and second order derivatives for a given layer are saved and passed on to previous layer computations to speed up the algorithm (i.e.

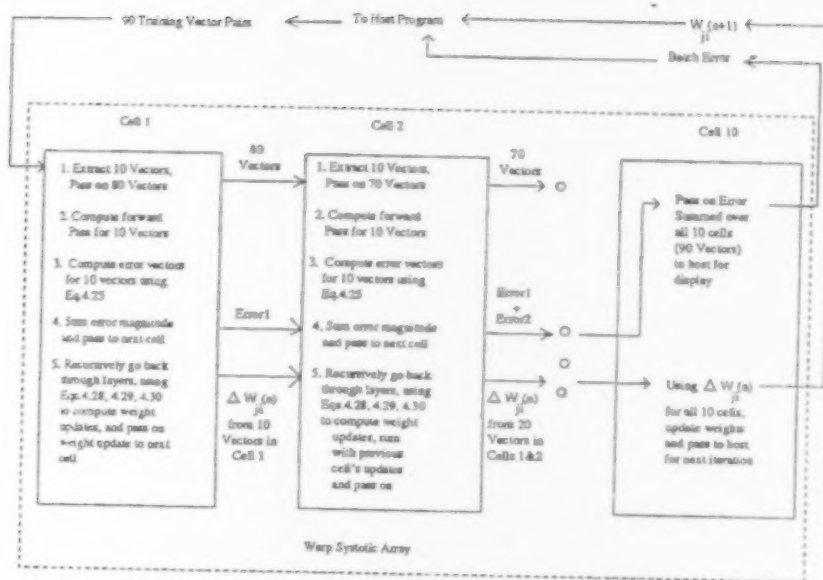


Figure 5.2: Description of an iteration of the batch backpropagation algorithm on the WARP systolic array. (Weights and interpolation tables assumed loaded in all cells).

$\varphi_j^p(n)$ in Eq. 4.39). The second order derivatives from cells 1-9 are also summed up and passed back to the host, completing the summation over the batch size. The user has the option to specify a threshold and use the second order derivatives to delete low-saliency weights at the host.

Chapter 6

Results

It is required to investigate the performance of the chosen preprocessing techniques and to determine the best overall method: preprocessor along with neural network architecture. Ideally, this would mean taking each preprocessor, extensively experimenting with its parameters, and training neural networks for each case to get a thorough feel for the performance. No doubt this technique would be very time-consuming. Due to the load limitations of the systems being used (*SUN Unix* for preprocessing, *WARP* for neural networks) and the requirements of other users, such an exhaustive approach is not possible. Rather, one can initially experiment between the various preprocessors on a difficult "trial" problem and then choose the one giving the best performance. The selected method is then thoroughly investigated, with many neural networks being training and optimized for it. This "divide-and-conquer" type of approach is chosen since it can significantly reduce the experimentation stage and still give very good results.

Before discussing the results, a few details about the signal database need to be mentioned. Specifically, there are 1,800 training, 100 validation, and 1,000 test examples generated for each of the 10 classes. The training and validation signals are stored in compressed form on a 1.6 Gigabyte hard disk, purchased specifically for contract purposes. The test signals are temporarily stored on an 1.0 Gigabyte magneto-optical disk. This means that overall, there are 18,000 training, 1,000 validation, and 10,000 test signals available to implement the classifiers with. Although these numbers may

seem large, considering the size of the problem, they are still well below some derived theoretical bounds (ie. Eq. 4.30, Eq. 4.31). However, practically, such large sizes are unrealistic since even this database is pushing system storage limits, and one has to make do with the available sizes.

To determine a good (ie. difficult) trial problem, a little experimentation is done with the preprocessors, using both 1-NN classifiers and neural networks. As a start, to get a feel for the problem, only two sizes of each PSD, periodogram, and bispectrum feature vectors are computed and classifiers trained to get preliminary results. (Note that the thorough approach would mean computing many different sizes of these vectors and fully testing each one. Just computing two sizes is in itself extremely time-consuming, with the runs lasting over several weeks). All preprocessors give the lowest recognition rates on the AM and FM classes. Also, as it turns out, most of the AM vectors tend to get misclassified as FM and vice versa. This problem is very prevalent when the voice signal is low (ie. silence). This is therefore a very good, yet compact, two-class trial problem against which to gauge the performance of the preprocessors. Using these two classes, each preprocessing technique is fully investigated using a wide range of input feature sizes. The results of these experiments are now presented.

6.1 Preprocessing

6.1.1 PSD Averaging

In PSD averaging, the number of points for the FFT is already predetermined by the sample size. Since there are roughly 8,250 sample points, an FFT size of 8,192 is chosen, the closest power of 2. A major issue with this approach is choosing the bandwidth to be processed. Signal specifications limit all signals to a maximum of 25kHz bandwidth, but even this value may not be required. Visually checking sample spectra of all the signal classes, a bandwidth of 10kHz is chosen. As it will be explained, this choice is good for both the AM/FM trial problem and the overall 10 class problem. Specifically, the analog modulations AM, USB, and LSB all occupy

under 6kHz. The digital modulations ASK, slow-rate FSK, and BPSK all have low enough data rates that most of their power spectrum falls within the stated 10kHz bandwidth about the carrier (ie. mainlobe and several sidelobes). QPSK can have a rate upto 9600 baud and therefore the mainlobe of the will take up almost all of the 10kHz bandwidth. FM and fast-rate FSK are specified to have bandwidths up to 25kHz, but it is noted that most of the FM signals' spectral power lies within 6kHz of the carrier (for the range of frequency deviations chosen, see Appendix B). Therefore it is really only fast-rate FSK which requires more than the allocated 10kHz to get an accurate picture of the power spectra. However, it is postulated that even within this range, the power spectra of fast-rate FSK is sufficiently distinct from all the other signal types' spectra so as not to cause major problems in discrimination. By choosing 10kHz, not only are the bandwidth requirements met for the AM/FM trail problem, but also for the whole 10 class problem.

10kHz is given by roughly 1092 bins in a 8192 point FFT, 75kHz sampling rate. This number is still too large to realistically use with classifiers (ie. computation time, memory requirements, training/test data sizes). Thus to reduce the input size further, groups of bins are averaged together (as suggested in Chapter 3). Note that is process is not equivalent to bandpass filtering since the magnitudes of the bins is taken in computing the periodogram (Eq. 3.9) and the average is therefore just a scalar sum. However, the averages will be referred to loosely as the outputs of "filters". Since the most critical region (with respect to discriminatory information) is close to the mean IF, the filters within a 6kHz range about the mean IF (ie. 12kHz-18kHz) are made narrower to give a finer resolution. The width of the filters within this bandwidth is constant and is narrower than that for filters outside this bandwidth (10kHz-12kHz and 18kHz-20kHz). There is no hard and fast rule which states that these heuristics are optimal in any sense. A lot of exhaustive experimentation would be required to find the optimal number of "filters" and their corresponding widths. Thoroughly investigating this route is not feasible. Rather, one can get a good feel for how fine to make the resolution by trying a few sample runs.

Filter sets of 10, 20, 30, 60 and 80 are used to preprocess the signals and the outputs used on both 1-NN and neural network classifiers. The "bandwidths" of

Filters	%AM	%FM	%Overall	%AM (log)	%FM (log)	%Overall (log)
10	89.33	86.33	87.33	94.33	88.00	91.17
20	93.33	93.33	93.33	95.67	93.67	94.67
30	90.33	94.00	92.17	94.00	97.00	95.50
60	90.67	91.33	91.00	92.50	94.50	93.50
80	86.00	90.33	88.17	90.00	92.00	91.00

Table 6.1: Results for 1-NN using PSD averaging for AM/FM problem

these filters varies a lot, with that for filters close to the IF being set to about two-thirds to half that of filters farther away (ie. outside 6kHz). It is obvious that the 80-filter set gives the best resolution, roughly 120 Hz for filters about the carrier. The finest filters for the 30-filter set are about 3 times as coarse, roughly 288 Hz, and those for the 10-filter set are the coarsest, giving a peak resolution of roughly 730 Hz. Table 6.1 tabulates the results of 1-NN runs with the accuracies being plotted in Figure 6.1. for the AM/FM problem. Limited runs with various size neural networks are also tried, and the performance is very close to the 1-NN of the corresponding input filter set (ie. same trend followed over the various filter sets). However, since training neural networks is much more time consuming than running 1-NN classifiers, not too many runs are done. Due to training time limitations, extensive neural network development has to be left until after the final preprocessor is chosen. Because the 1-NN's performance on a particular filter set is also highly indicative of the performance of various sized neural networks (typically within $\pm 0.4\%$), it is used to judge the filter sets to choose the best one. The peak performance occurs for the 30-filter set, giving roughly 95.50% for the independant test set. It is clear from that graph that adding more filters does not increase performance and that a good set should have between 20 to 30 filters. A significant improvement is seen when the scaled-log of the filter outputs is used.

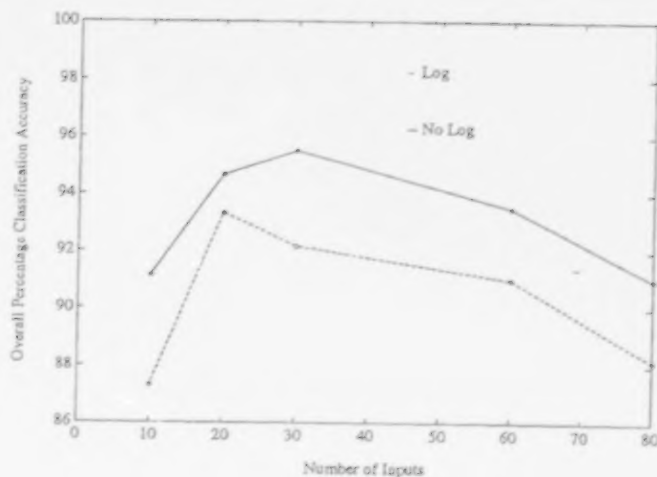


Figure 6.1: Performance of 1-NN for AM/FM Problem, PSD Averaging

6.1.2 Welch's Modified Periodogram

Unlike the PSD averaging method, the Welch method does allow flexibility in the FFT size chosen. This technique basically computes smaller FFT's and then averages them to yield an averaged estimate of the power spectrum. In this approach, the smaller FFT size, M (Eq. 3.20), controls the maximum resolution of the technique. Using the same reasoning as in the PSD averaging method, a 10kHz bandwidth about the carrier is chosen. Thus, Welch periodogram estimates for the signals are computed and the points in the M -point estimate corresponding to the desired bandwidth are extracted. Note that unlike the PSD averaging technique, the resolution is fixed across the whole bandwidth. Averaging of bins can also be done here but since the M values can be chosen to give reasonable dimensionalities, this is not done. Table 6.2 and Figure 6.2 summarize the 1-NN results for the AM/FM problem using the periodogram preprocessor (both scaled-log and regular input). From these values, $M = 256$ (ie. 70 input parameters) with scaled-log input gives the best recognition. Figures 6.3 and 6.4 show the improvement obtained by taking the log of the final

K	M	Bins (10 kHz)	AM	FM	Overall	AM, log	FM, log	Overall, log
128	64	18	83.67	85.33	84.50	94.00	90.33	92.17
64	128	35	91.67	94.33	93.00	93.33	96.00	94.67
32	256	70	95.00	100.00	97.50	98.67	100	99.33
16	512	135	-	-	-	94.33	95.00	94.67

Table 6.2: Results for 1-NN using periodogram for AM/FM problem

estimate. The scaled-log of the estimate "brings out" the sideband information, much more so than the straight Welch estimate. It is this information about the carrier that is critical for discrimination. The performance of this technique is also significantly better than that of PSD averaging for a roughly equivalent number of filters. This can be explained to an extent by the fact that since the signals are random (ie. voice or random bit stream modulation), the time varying nature of the signals is important. The Welch periodogram accounts for this by essentially averaging a whole set of FFT's of at different time intervals in the signal. This gives a good estimate of the power spectrum if "enough" averages are taken. The PSD averaging technique rather takes a one-snapshot look at the power spectrum and therefore has increased variability leading to a decline in accuracy. One can get a sense of difference between the two techniques by looking at comparable-size input vectors from the two methods. For example, Figures 6.5 and 6.6 plot the preprocessed output (10kHz) for the same AM signal. The PSD-averaged plot is that of an 80-filter set and the Welch periodogram plot for a 70 input case (ie. $M = 256$). There is a major difference between the outputs of the two techniques, with the PSD averaging showing a lot more variability, especially between successive points in the output. This trend is also observed with PSD output for the remaining 8 classes: the outputs of the Welch periodograms show less "deviation" for different samples than do the PSD averaged outputs. Note that a 256-point Hamming window is used to weight each estimate, and the windows are not overlapped. Some runs with overlapped windows show no reduction in the error rates and therefore this option is not considered further.

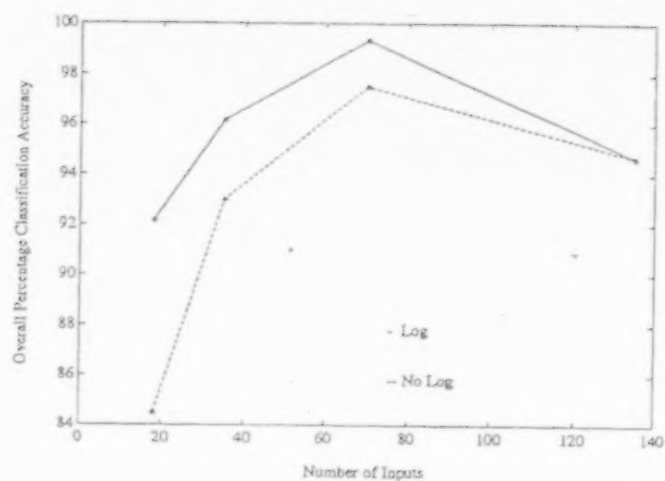


Figure 6.2: Performance of 1-NN for AM/FM Problem, Welch Periodogram

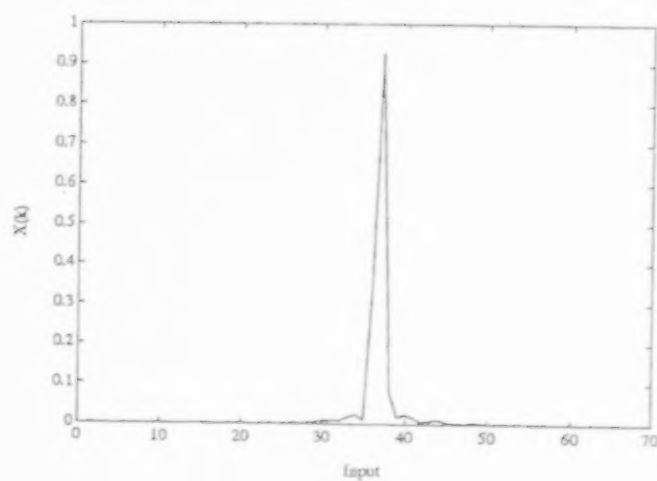


Figure 6.3: Welch periodogram for FM signal, $M = 256$, 70 bins (10kHz)

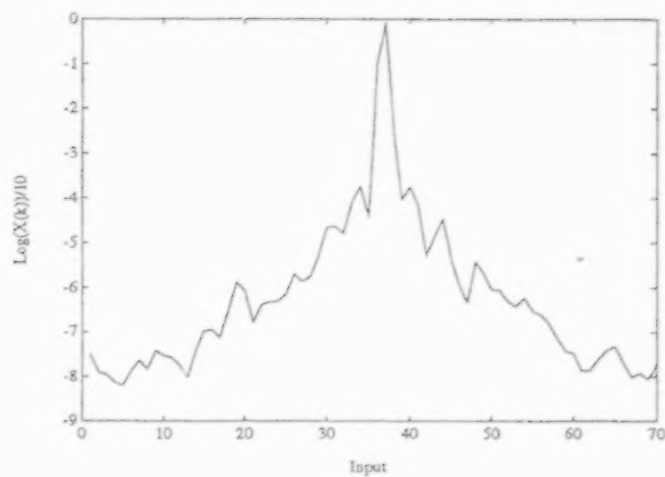


Figure 6.4: Scaled-log of Welch periodogram for FM signal, $M = 256$, 70 bins (10kHz)

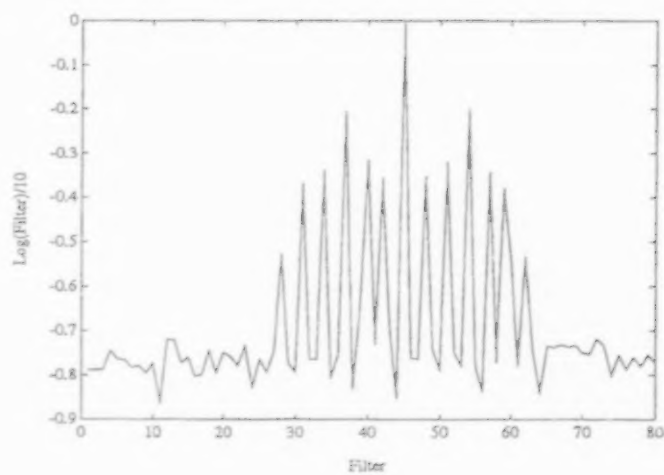


Figure 6.5: Sample PSD averaged output for AM signal, 80-filter set chosen to represent 10kHz bandwidth.

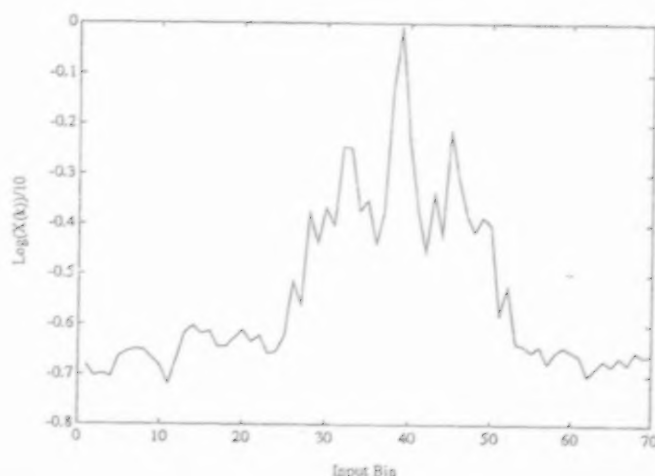


Figure 6.6: Sample Welch periodogram output for AM signal, $K = 32$, $M = 256$, 70 bins chosen to represent 10kHz bandwidth

6.1.3 Bispectrum

A problem with both the PSD averaging and Welch methods is that only the relative phase information of the signal is extracted. This can be explained by the single time delay in the autocorrelation sequence. Higher order spectra can extract exact phase, and this sort of information can hold valuable discriminatory power, especially if the power spectrum of two or more signal types is very similar. It is hoped that the bispectrum method captures more of the phase-related information through the use of third order moment sequences. Using the method suggested by Nikias for bispectrum estimation (see Chapter 3), one can vary the FFT size and hence the resolution in the bispectral domain. Note, however, that with no sort of averaging done on the outputs the dimensionalities increase geometrically with increasing M (Eq. 3.47), as shown in Figure 6.9. For example, for $M = 256$, the number of features generated by this approach reaches the same order as the number of training vectors over all 10 classes, $10^4!$. These large dimensionalities require much larger training sets to give good

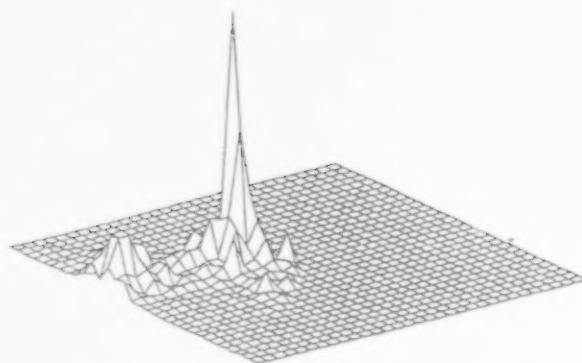


Figure 6.7: Sample bispectrum of an AM signal, FFT size, $M = 64$, 272 points overall

performance. This problem will definitely restrict the sizes of bispectra that can be used since the computing systems cannot handle input dimensionalities over roughly 600. As a result, only FFT sizes of 32 and 64 points can be used, giving bispectral vectors of length 72 and 272 respectively (see Figures 6.7 and 6.8 for sample plots). In addition, as mentioned previously (Chapter 3), since the information in the bispectral domain is not localized to select regions, one cannot discard certain inputs as is done in the previous two preprocessing methods (ie. bandwidth assumption).

Table 6.3 tabulates the results with a 1-NN for the AM/FM problem using the two chosen bispectrum sizes (optimal windowing, magnitudes of bispectral bins taken). The results do not improve when scaled-log inputs are used and they are significantly below those obtained by either of the two previous methods. This is most probably due the fact that the dimensionalities are significantly higher, and the resolution in the bispectral domain is coarser. For example, for the Welch method, peak performance is obtained for an FFT size of 256, 146 Hz resolution, whereas the 64-point bispectrum can only give about 585 Hz resolution. Attempts are made to use larger

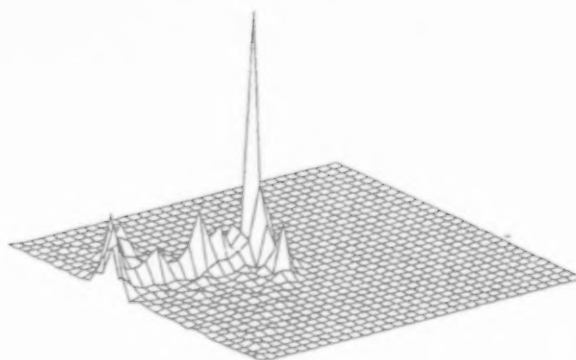


Figure 6.8: Sample bispectrum of an FM signal, FFT size, $M = 64$, 272 points overall

K	M	Inputs	% AM	% FM	% Overall
256	32	72	84.59	86.05	85.32
128	64	256	86.33	85.67	86.00

Table 6.3: Results for 1-NN using bispectrum for AM/FM problem

bispectra ($M = 128$, 1052 points), and average groups of bins together to reduce input dimensionalities. However, the results from these runs are even worse than those obtained by the smaller bispectrum sizes. It is obvious that the curse of dimensionality problem is very serious here. To equal or outdo the performance of the PSD averaging and Welch methods using bispectrum, there has to be a tremendous increase in both the number of training samples and the computing/memory resources available. For the given database size, the bispectrum cannot perform effectively and therefore is not investigated any further.

In light of the above experiments, the log-scaled Welch periodogram with FFT

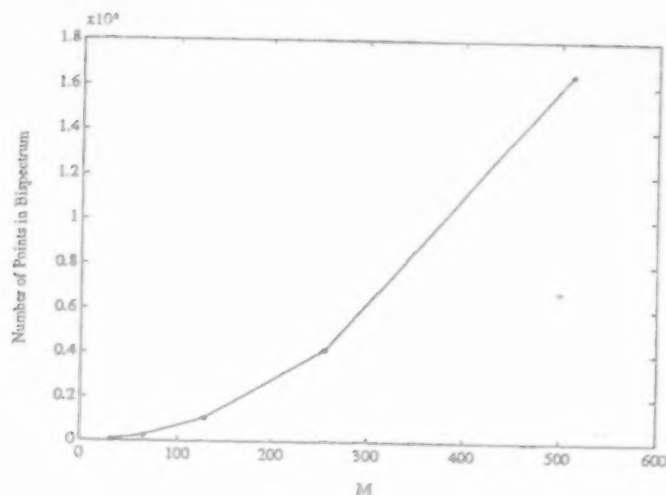


Figure 6.9: Plot of bispectral parameter count vs. FFT size, M

size 256 is chosen as the final preprocessor for the system. This method yields the best results for the AM/FM trail problem and the input dimensionality is within the range of the computing resources. For this method, the full 10-class problem is investigated with a range of backpropagation networks being implemented. Figures 6.10- 6.14 plot the mean input vectors for this approach for all 10 classes, averaged over 1,000 samples (note the difficulty in AM/FM discrimination). It should be noted that the plots are much smoother than those for individual samples, and also, the bandwidth for the signals seems to be wider. This is because the averaging is done over many samples, and the IF varies (see Appendix B). Even though these averaged plots look quite different, it is deceptive to think that discrimination is easy. The plots represent averaged values and individual plots show much more variability (and hence similarity). The scaled-log enhances the difference between the classes, since both the power and shape are being accounted for.

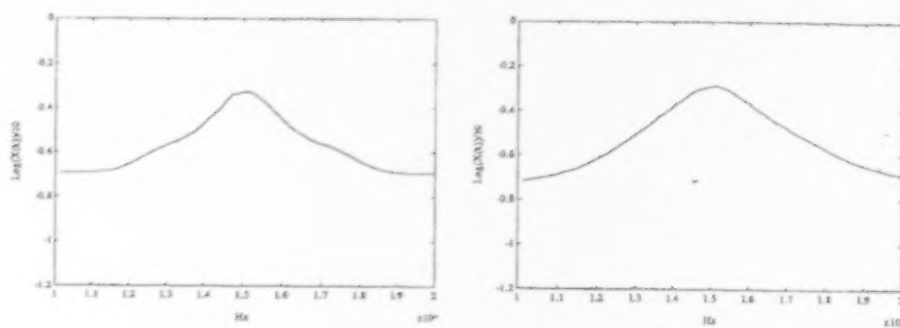


Figure 6.10: Averaged Welch Periodograms, $M = 256$. AM (left), FM (right)

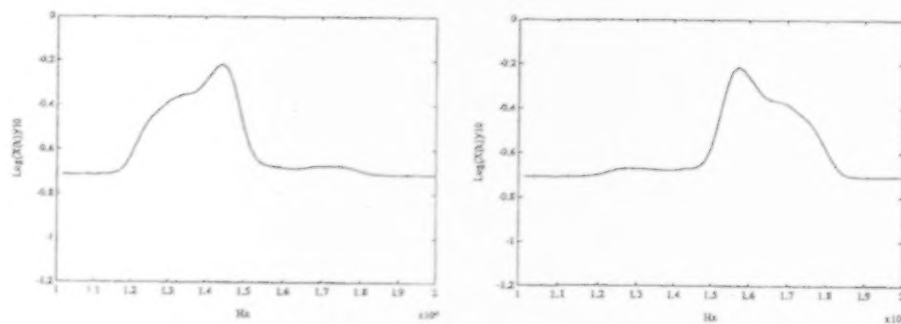


Figure 6.11: Averaged Welch Periodograms, $M = 256$. USB (left), LSB (right)

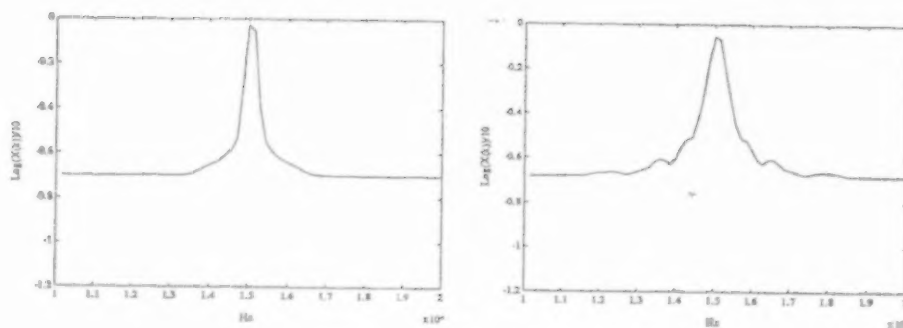


Figure 6.12: Averaged Welch Periodograms, $M = 256$. ASK (left), BSPK (right)

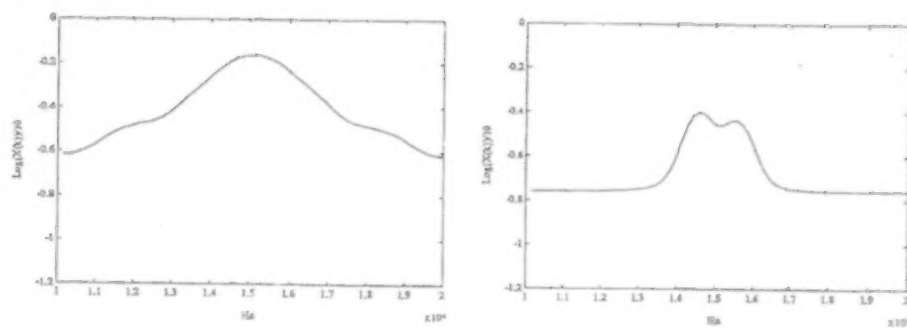


Figure 6.13: Averaged Welch Periodograms, $M = 256$. QPSK (left), slow-rate FSK (right)

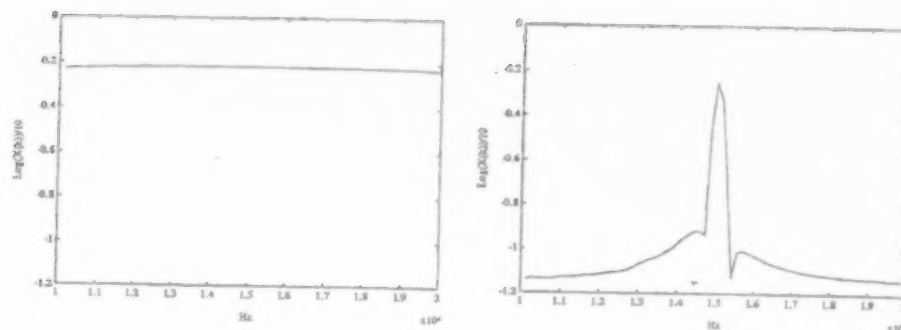


Figure 6.14: Averaged Welch Periodograms, $M = 256$. fast-rate FSK (left), CW (right)

6.2 Classifiers

Since the final preprocessor has been chosen, experiments are done with both k-NN classifiers and single/double hidden layer neural networks. The performance of the two classifiers is to be compared. The results of these experiments are now presented.

6.2.1 K-Nearest Neighbour Classifier

Using the Welch periodogram method, $M = 256$, k-NN runs are done for the full 10-class problem and varying values of k . The Euclidean-distance metric is used to avoid having to perform parameter estimation. It is postulated (section 7.1) that the signal spectra are sufficiently distinct over a 10kHz bandwidth to use this value. Nevertheless, it can also be asked whether this bandwidth is large enough or conversely whether the signals are sufficiently distinct over an even smaller bandwidth to allow acceptable performance? If so, this would reduce the number of inputs further. Runs with bandwidths larger than 10kHz give reduced accuracies, indicating too many features. Thus smaller sets should be tried and as an attempt, 6kHz and 5kHz bandwidths about the mean IF are extracted, reducing the number of inputs from 70 to 42 and 35 respectively. Figure 6.15 plots the performance of the k-NN for the

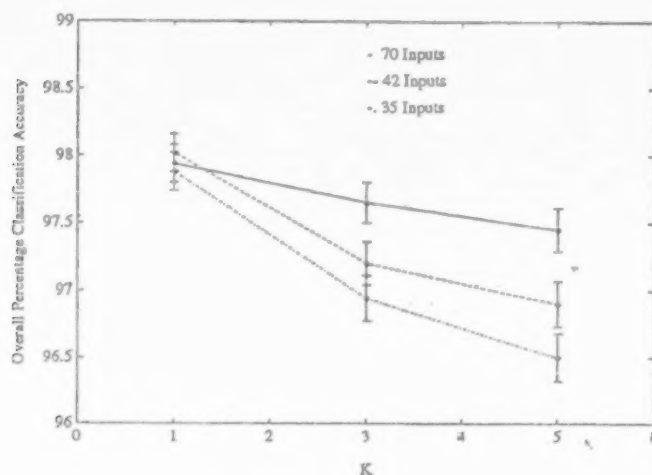


Figure 6.15: Performance of k -NN for Welch Periodogram, $M = 256$, 10 classes

chosen bandwidths and increasing k . The error bars indicate one standard deviation in the estimate—in other words, roughly two thirds probability that the estimates will lie between these bars. These results are quite good, indicating good separability between the classes in the input space, and in a sense justify the choice of the preprocessor. Note that the 1-NN (Table 6.4) performs the best for both input sizes, indicating the severity of the undersampling in the high-dimensional input space being done by the training dataset. In other words, as the k nearest neighbours of a test point are computed, the volume in the hyperspace, $A(k, N, \underline{x})$ in Eq. 4.3, has to become disproportionately large to accomodate them due to the sparse sampling. This will probably give bad results especially around boundaries between classes. Such an effect is observed in the results as increasing k worsens the accuracies for the difficult AM and FM classes more so than other classes. It is surprising to see that using a bandwidth smaller than 10kHz causes little change in the *overall performance*. Specifically, for both 42 and 35 input cases, there is an improvement in the AM/FM problem with slight decreases in the other classes. (Note that another difficult problem is the BPSK/FM problem, where most of the incorrect BPSK vectors are being

B	AM	FM	ASK	QPSK	USB	LSB	FSK1	FSK2	BPSK	CW	Overall
70	96.5	96.3	96.4	99.4	99.7	99.8	98.7	99.5	93.1	100	97.94
42	97.0	96.8	96.0	98.7	100	99.8	98.9	99.2	93.8	100	98.02
35	97.1	96.8	95.9	97.2	100	99.7	98.8	99.1	94.3	100	97.88

Table 6.4: Percentage accuracies of 1-NN for 10-classes, 1,000 test per class, B bins extracted

classified as FM but not vice versa). These results seem to indicate that there may be too many parameters being extracted by using a 10kHz bandwidth, slowing down the classifier and hindering performance. These differing input sizes should be used with the neural networks to see whether or not similar behaviour holds. Due to computer-time limitations, only the 10kHz and 6kHz inputs are chosen since they have better overall performance than the 5kHz input. It should be mentioned that runs are also done with bandwidths less than 5kHz but the results are significantly below the above values.

6.2.2 Backpropagation Neural Network Classifier

Both single and double hidden layer backpropagation networks are trained for the 70 and 42 input Welch periodograms. With a single hidden layer, the major parameter to be determined is the size of this layer. The input size is already determined by the preprocessing, and the output size by the number of classes [20]. There are no fixed rules stating how large this layer should be since it is very data-dependant and ties in directly to the separability between the classes. A reliable heuristic states that the hidden layer should have at least as many neurons as inputs, maybe upto two to three times this value. Typically, more hidden neurons are required to distinguish between close classes (AM/FM). Nevertheless, a simple experimental technique can be used to find an "optimal" hidden layer size. Basically, different networks are trained with increasing number of hidden neurons and the error rates compared. If the number of hidden neurons is too low, then the error rate will be high because the network cannot model the decision surfaces adequately. This means that the number of hidden

H	W	AM	FM	ASK	QPSK	SSBU	SSBL	FSK1	FSK2	BPSK	CW	Overall
80	4250	94.2	92.9	96.2	99.3	99.2	99.6	98.5	100	98.2	100	97.81
120	6370	95.5	96.0	96.2	99.6	98.0	100	98.4	100	98.2	100	98.37
140	7430	96.0	95.2	95.7	99.5	99.4	99.7	98.7	100	98.3	100	98.25
200	10610	96.6	94.6	96.0	98.9	99.3	99.5	97.9	100	98.2	100	98.10

Table 6.5: Performance of fully-connected backpropagation network with single hidden layer of H hidden neurons, W weights, 42 inputs (6kHz). Results averaged over 3 independent training runs.

H	W	AM	FM	ASK	QPSK	SSBU	SSBL	FSK1	FSK2	BPSK	CW	Overall
120	9730	98.6	94.9	95.9	99.8	99.6	100	98.9	100	97.8	100	98.35
130	10540	96.7	96.3	96.1	99.8	99.5	99.6	98.8	100	98.4	100	98.50
140	11350	96.0	94.8	96.5	99.5	99.4	99.8	98.2	100	97.6	100	98.18
200	16210	97.1	89.9	96.1	99.1	99.0	99.2	97.3	100	96.8	100	97.45

Table 6.6: Performance of fully-connected backpropagation network with single hidden layer of H hidden neurons, W weights, 70 inputs (10kHz). Results averaged over 3 independent training runs.

neurons has to be increased. As long as the error rate keeps declining, the hidden layer is increased and network training continued. If this value is increased too far, then the error rate will begin to increase again since the network is not generalizing properly. Therefore by trying a wide range of sizes, a "peak" in the accuracy can be found.

Tables 6.5 and 6.6 summarize the results of single-hidden layer networks and Figure 6.16 plots the overall accuracies, error bars for standard deviation shown. It should be mentioned that experimentation shows that small learning rate values (≤ 0.05) using the fan-in heuristic (Section 4.2.2) give the lowest mean-squared error, MSE, on the training set.

Like the 1-NN, these results are very good. First of all, the overall performance is very close to that of the 1-NN and in some cases better. Increasing the hidden layer size shows a significant improvement in the especially the AM/FM accuracies upto the "optimum" size. The most intriguing result is that the peak performance

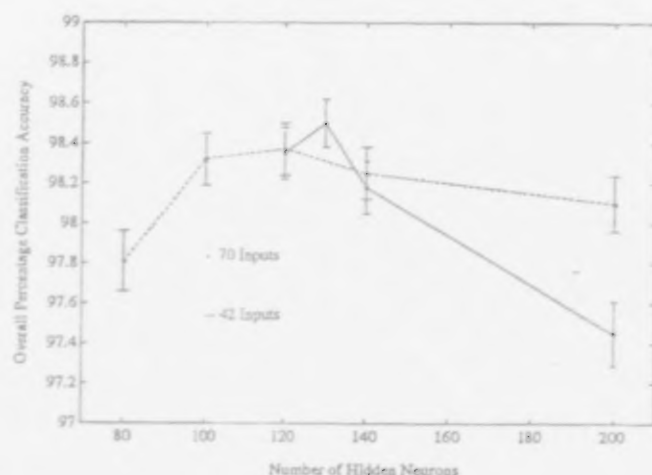


Figure 6.16: Fully-connected single hidden layer network performance, Welch periodogram, $M = 256$

of the smaller net is very close, considering the variance in the estimates, to that of the larger net. Further, both nets seem to peak around roughly 120 hidden units. This result indicates that the added inputs with the 10kHz bandwidth input are not improving discrimination significantly. Considering that there is about a 60% increase in the number of weights for the "optimum" network for the larger input size over the smaller one, the slight gain in performance may not be very worthwhile. Note that for both input sizes, the performance begins to fall off rapidly with the increase in hidden layer size, with the difficult classes and larger input sizes declining more. This is probably due to the decrease of the ratio between the number of training vectors to weights. Experimentation shows, for single- hidden layer networks, the above ratio should be at least greater than about 1.5 to get results over all classes close to those of the 1-NN. Typical nets are trained for at least 5,000 iterations, with the validation set being tested about every 400 iterations. Larger net sizes tend to give slightly lower MSE's due to better interpolation abilities with more weights, but require more iterations. Still, all of the smaller nets give MSE's less than twice those for the larger

nets. In most cases, the training stage (ie. with intermittent validation being done) lasts roughly 20 hours. Larger nets can take up to twice this time to train if problems with local minima are encountered. It is clear that despite the speed of the WARP, training time is a major bottleneck.

Double-hidden layer networks are also trained for the Welch periodogram preprocessor. Since there are two major parameters to vary now, determining a good network size requires significantly more experimentation. The added layer gives better discriminatory ability to the network since two hidden layers can model any arbitrary decision region, given enough neurons of course. The approach followed in investigating the performance of two-hidden layer networks is somewhat similar to that for single-hidden layer networks. Specifically, the size of the second hidden layer is held fixed and that of the first varied to get a wide range on the performance. Next, the size of the second hidden layer is changed and the procedure repeated until enough sizes of the second hidden layer have been tested. No doubt that the overall training time is much larger now.

Many different networks are trained for the two-hidden layer case, and some of the results are now presented. Figures 6.17 and 6.18 plot overall performances of both input sizes for various sizes of hidden layers. Results show that both input sizes give best performance for roughly the same number of neurons in the second hidden layer, 30. Nevertheless, the performance over a very wide range of network sizes is very close and this is indicated by the bars on the estimates. Also, tests with several networks shows that there is high variability, upto $\pm 0.2\%$ overall rate, between independent training runs with a fixed network size. This makes it difficult to draw very exact conclusions from the values. Tables 6.7 and 6.8 tabulate the results for these networks for 30 neurons in the second hidden layer. For the overall rate, the larger input size does slightly better, although the discrepancy between the AM and FM rates is larger than for the smaller input size. Compared to the results of the single hidden layer, the double hidden layer does not necessarily perform better. For example, the "optimum" single-hidden layer network for the 70 input size gives an overall accuracy of 98.50% (Table 6.6). This is slightly below that given by the best double-hidden layer, 98.61%, and within a standard deviation of

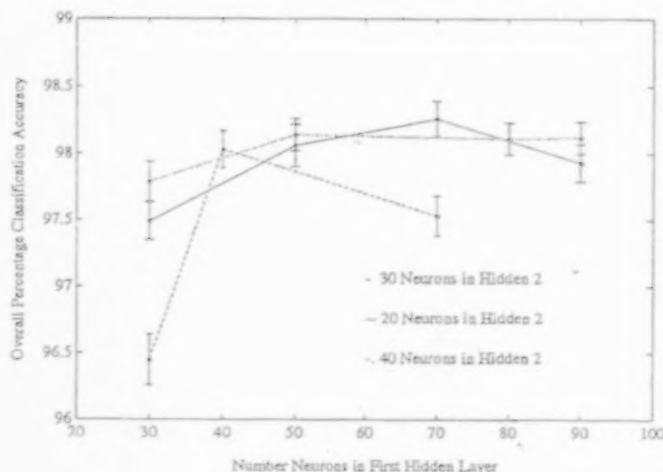


Figure 6.17: Fully-connected double hidden layer network performance for 42 inputs, Welch periodogram, $M = 256$

the error rate estimate, approximately 0.12%. For the smaller input size, the “best” single-hidden layer result, 98.37% (Table 6.5), is slightly better than that given by the corresponding double hidden layer size, 98.26%, but again within a single standard deviation of the estimates. These results hint that there is good separability between the classes in the chosen Welch feature space and that added capabilities of double hidden layer networks are not required since decision regions are being modelled just as well by a single hidden layer.

Comparing the results obtained with the backpropagation networks to those given by the 1-NN, it can be said that the “optimum” backpropagation networks give lower error rates on the test set over most classes and a lower overall error figure. The drawback of using the neural networks is that the training time on even the WARP computer is at least 3 times that required to compute 1-nearest neighbours for all 10,000 test points. However, once a network has been trained, forward passes for strict classification are tremendously faster on the WARP, giving close to real-time speed. Although techniques to shrink the required training set size for the k-NN

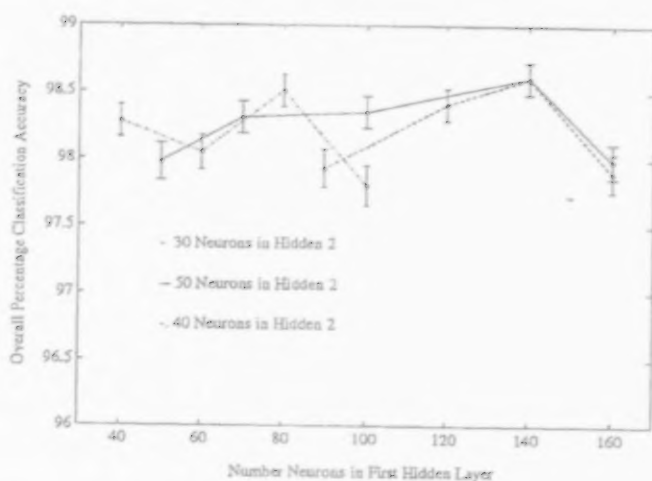


Figure 6.18: Fully-connected double hidden layer network performance for 70 inputs, Welch periodogram, $M = 256$

H1	W	AM	FM	ASK	QPSK	USB	LSB	FSK1	FSK2	BPSK	CW	Overall
30	2530	93.2	93.5	95.2	99.2	99.2	98.9	98.0	99.9	97.8	100	97.49
50	3990	95.1	93.8	96.3	99.6	99.5	99.6	98.7	100	98.0	100	98.06
70	5450	95.2	96.1	96.3	99.5	99.5	99.6	98.3	100	98.1	100	98.26
90	6910	95.7	93.2	95.9	99.2	99.5	99.4	98.3	100	98.1	100	97.93

Table 6.7: Performance of fully-connected, double-hidden layer backpropagation network, H1 hidden neurons in first hidden layer, W weights, 30 neurons in second hidden layer, 42 inputs. Results averaged over 3 independent training runs.

H1	W	AM	FM	ASK	QPSK	USB	LSB	FSK1	FSK2	BPSK	CW	Overall
50	5390	96.0	92.4	96.8	99.5	99.3	99.1	99.1	100	97.6	100	97.98
70	7410	97.2	94.7	97.0	99.7	99.7	99.6	98.7	100	96.5	100	98.31
100	10440	97.5	95.2	96.8	99.7	99.2	99.5	98.4	100	97.2	100	98.35
140	14480	97.8	95.7	96.3	99.6	99.6	99.9	99.0	100	98.2	100	98.61
160	16500	97.5	95.0	96.3	99.5	99.4	99.9	98.8	100	97.7	100	97.89

Table 6.8: Performance of fully-connected, double-hidden layer backpropagation network, H1 hidden neurons in first hidden layer, W weights, 30 neurons in second hidden layer, 70 inputs. Results averaged over 3 independent training runs.

(and hence increase speed) do exist, the large number of classes present will probably not give significant reduction. Thus for classification purposes, the neural network requires less computing resources (after training) than the k-NN. As a result, the neural network approach is much more attractive for application in a real-world AMR system.

From the results for both single and double hidden layer networks, it is clear that the AM, FM and BPSK classes have the lowest performance. Although the recognition rate is still very good for these classes, over 90% in most cases, it is roughly 2% below that of other classes. It is postulated that higher order moment information contained in the bispectrum will give added information to improve discrimination on these classes, but earlier attempts with the bispectrum have not been fruitful. As a compromise, some higher order moment information can be added by simply computing these moments. Such an attempt is made with the skew and kurtosis. Specifically, the 70 and 42 Welch periodogram input vectors are augmented with the skew and kurtosis of the *time* sample sequence. In other words, now the input dimensionalities become 72 and 44, respectively. The skew is the third-order moment and is defined as [22]:

$$\text{skew}(x_1, x_2, \dots, x_N) = \frac{1}{N} \sum_{i=1}^N \left(\frac{x_i - \mu_x}{\sigma_x} \right)^3 \quad (6.1)$$

and the kurtosis is the fourth-order moment, given by:

$$\text{kurtosis}(x_1, x_2, \dots, x_N) = \left\{ \frac{1}{N} \sum_{i=1}^N \left(\frac{x_i - \mu_x}{\sigma_x} \right)^4 \right\} - 3 \quad (6.2)$$

"The skew characterizes the shape of the distribution, and the kurtosis measures the relative peakedness or flatness of a distribution relative to a Gaussian" [22]. For example, Figures 6.19 and 6.20 show very typical plots of AM and FM signal histograms. There is a marked difference between the shape of the two, and this should be captured by using the skew and kurtosis. To see this, a skew vs. kurtosis plot can be done to see the distribution of the AM and FM signals in this domain. Figure 6.21 shows this plot for 300 AM and FM samples. It is clear that the clustering of the two classes is well separated in this domain, and therefore this will provide added discriminatory information to the classifier. Runs with the 1-NN for both augmented input sizes show roughly a 2% improvement in the difficult classes-AM, FM and BPSK. All other classes have the same recognition rates as with the non-augmented feature set, and thus the overall recognition rate increases slightly, close to 99%. A few neural networks of varying sizes are also trained for this new set and similar properties hold.

6.2.3 Optimal Brain Damage

The OBD algorithm is used to optimize some of the trained networks. Since this procedure is very time consuming, only the networks with better performance are pruned. For the single hidden layer case, OBD is done for the 42 input network with 120 hidden neurons and for the 70 input network with 120 and 130 neurons. Sample results for these runs are presented in Tables 6.9- 6.11 and Figures 6.22- 6.24. These results indicate that much more of the larger networks can be deleted (relatively), with the performance holding within the bounds of the standard deviation. This agrees with the postulation earlier that the full 70 input case may have too many features. This is further supported by the plots in Figures 6.25 and 6.26. In these plots the magnitudes of the weights originating from each *input* node are summed and plotted against the node itself for significantly-pruned 70 input networks. Loosely,

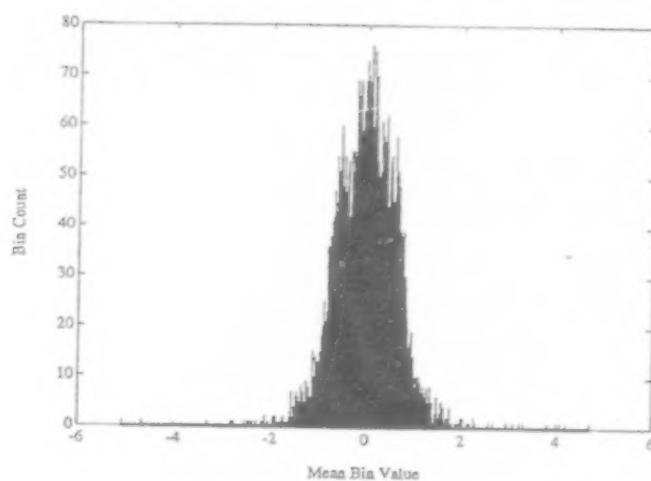


Figure 6.19: 1000-bin histogram plot of an AM signal

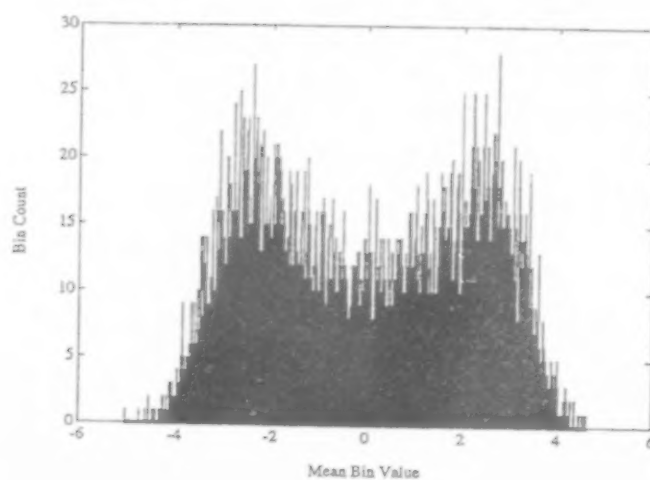


Figure 6.20: 1000-bin histogram plot of an FM signal

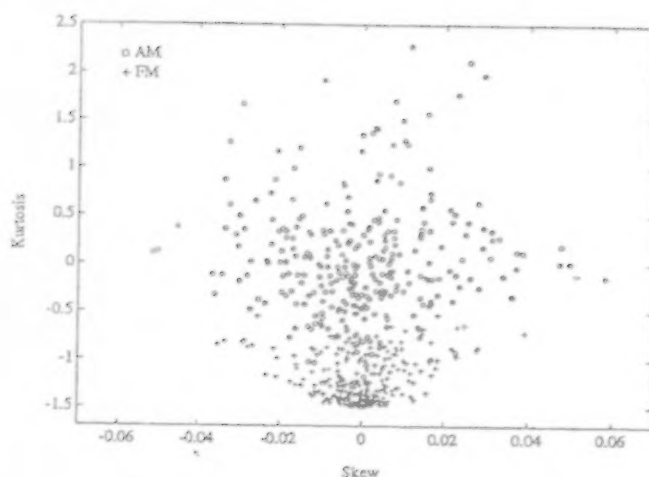


Figure 6.21: Kurtosis vs. skew plot, AM and FM classes, 300 samples each

such a cumulative figure gives a sense of “importance” to each weight. Remarkably, the inputs close to the edges have considerably less cumulative magnitude than those near the middle. This confirms that the inputs close to the IF are more important, but still does not tell exactly how many inputs are required to get optimal recognition.

In some cases, the overall performance goes up slightly as weights are deleted.

W	AM	FM	ASK	QPSK	SSBU	SSBL	FSK1	FSK2	BPSK	CW	Overall
6370*	95.5	96.0	96.2	99.6	98.0	100	98.4	100	98.2	100	98.37
5590	95.7	96.3	96.2	99.6	99.8	99.9	98.4	100	98.2	100	97.38
5140	95.9	95.7	96.2	99.4	99.7	99.9	98.4	100	98.2	100	98.24
4256	95.7	96.0	96.2	99.6	99.8	99.9	98.4	100	98.2	100	98.38
3822	82.0	97.2	97.5	99.7	99.2	99.3	98.0	100	98.1	100	97.09

Table 6.9: Sample OBD run for 42 inputs, 120 hidden. W is number of weights, and * indicates results of fully-connected network.

<i>W</i>	AM	FM	ASK	QPSK	SSBU	SSBL	FSK1	FSK2	BPSK	CW	Overall
10540*	96.7	96.3	96.1	99.8	99.5	99.6	98.8	100	98.4	100	98.50
9698	97.1	96.1	97.1	99.9	99.5	99.6	99.0	100	99.6	100	98.49
8792	97.1	96.2	97.1	99.9	99.5	99.6	98.9	100	96.7	100	98.50
5947	96.9	95.7	97.1	99.9	99.5	99.6	98.8	100	97.4	100	98.49
3786	97.5	96.0	97.0	99.8	99.6	99.9	99.0	100	97.4	100	98.62
2740	92.0	97.2	97.1	99.6	99.5	99.9	99.2	100	95.9	100	98.14

Table 6.10: Sample OBD run for 70 inputs, 130 hidden. *W* is number of weights, and * indicates results of fully-connected network.

<i>W</i>	AM	FM	ASK	QPSK	SSBU	SSBL	FSK1	FSK2	BPSK	CW	Overall
9730*	98.6	94.9	95.9	99.8	99.6	100	98.9	100	97.8	100	98.35
9264	96.8	95.2	95.7	99.9	99.6	100	98.9	100	97.9	100	98.40
8848	96.7	95.0	95.6	99.9	99.6	100	98.9	100	97.8	100	98.35
7608	96.6	95.0	95.7	99.9	99.6	100	98.7	100	97.9	100	98.34
6331	95.8	95.2	95.8	99.9	99.6	100	99.0	100	97.8	100	97.31
5326	95.8	95.1	95.9	99.8	99.4	100	99.0	100	98.0	100	98.30
2965	96.0	94.4	95.6	98.2	99.4	99.6	98.1	100	97.6	100	97.89

Table 6.11: Sample OBD run for 70 inputs, 120 hidden. *W* is number of weights, and * indicates results of fully-connected network.

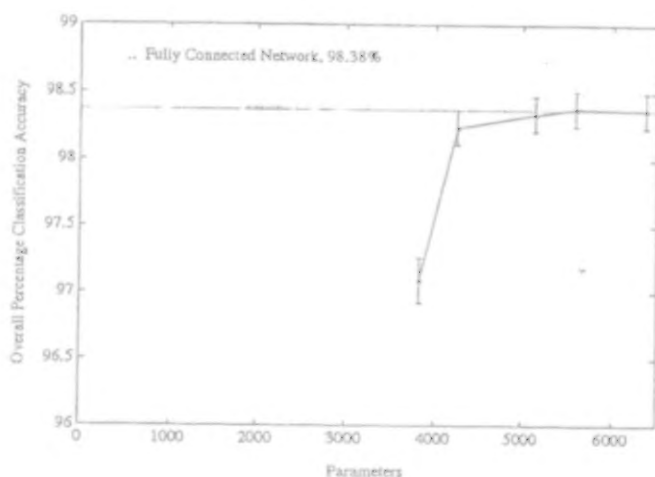


Figure 6.22: OBD for 42 inputs, 120 hidden. Fully connected network has 6,370 weights.

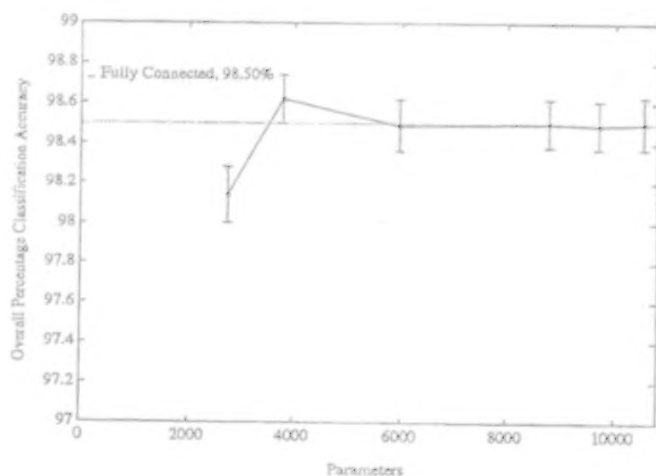


Figure 6.23: OBD for 70 inputs, 130 hidden. Fully connected network has 10,540 weights.

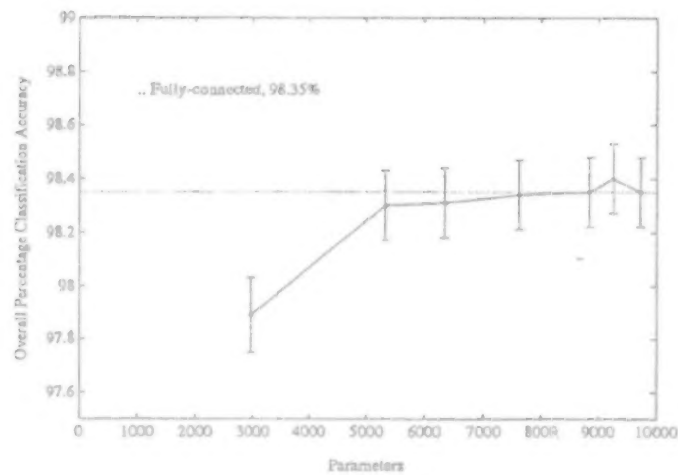


Figure 6.24: OBD for 70 inputs, 120 hidden. Fully connected network has 9,730 weights.

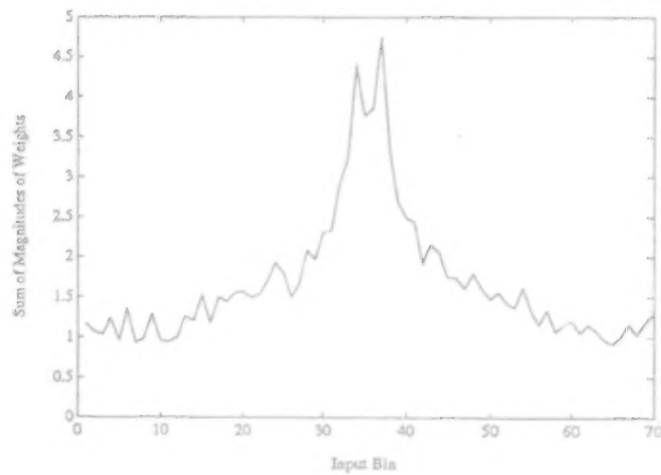


Figure 6.25: Cumulative weight magnitude vs. weight, 70 inputs, 120 hidden, 6331 weights (ie. 35% deletion).

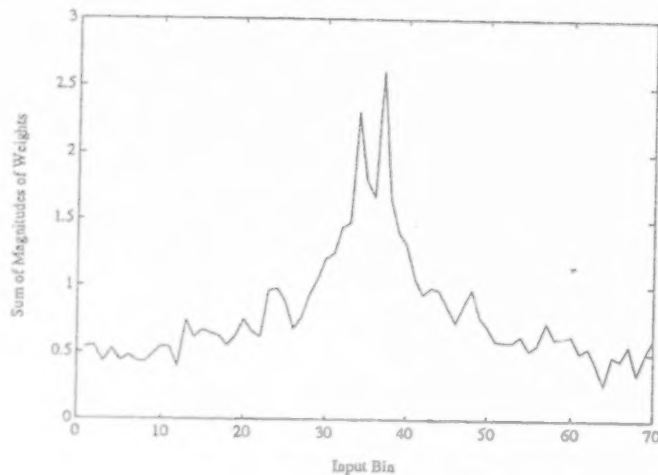


Figure 6.26: Cumulative weight magnitude vs. weight, 70 inputs, 130 hidden, 3786 weights (ie. 64% deletion).

This means that the initial fully connected network has too many weights and is not generalizing properly. OBD actually improves the generalization by reducing the overfitting. Despite the fact that the performance does eventually begin to decline as the deletion is increased, these results are still very surprising. For example, in all cases, deleting over half the weights still keeps overall accuracies over 97%. The decline in accuracy is probably due to several reasons. First of all, too many weights may be getting zeroed. Secondly, the discarded off diagonal elements of the Hessian matrix (Eq. 4.33) are probably becoming proportionally larger. Finally, deleting a large number of parameters may give higher order terms more significance in Eq. 4.32 and hence the assumptions made to derive the OBD equations would no longer be valid.

Results show that for both 42 and 70-input networks most saliencies are very small. A very indicative plot for both input sizes is shown in Figure 6.27, where a sample "cumulative saliency distribution" plot is shown for a 70-input, 130-hidden layer network. This graph is generated by first computing a 500-bin histogram on

the saliency values to get a rough idea of the distribution. Then a running sum of the histogram is computed to get a "cumulative" plot. For this particular network, the saliency values range from order 10^{-3} to 10^3 . This is a very wide spread, but well over 90% of the networks weights have extremely low saliency values. It is deceptive, however, to think that such a large portion of the network is "redundant". Rather, since the single-precision arithmetic on the WARP forces the use of small magnitudes, large double derivatives pertaining to non-redundant parameters can be masked by significantly small weight values, Eq. 4.35.

In actually running OBD, histograms are used to determine thresholds to attain certain deletion percentages. Once the required weights are zeroed, the network is retrained until improvement on the validation set levels off (step 5, Section 4.2.3). The number of iterations required is dependant on the percentage deletion, and like the MSE, increases as the percentage deletion is upped. In all runs, a minimum of 1000 iterations and 4 runs on the validation set are required. A larger MSE is to be expected since the reduced parameter set is supposed to generalize better and therefore not fit the training set that closely. Nevertheless, over all the OBD runs done, the MSE stays under twice the value of the original, fully-connected network.

OBD is also applied to some double-hidden networks, but since the procedure is very tedious only networks giving better performance are considered. Specifically, the 42 input network, with 70 first hidden layer and 30 second hidden layer neurons and the 70 input network, with 140 first hidden layer and 30 second hidden layer neurons are tried. Figures 6.28 and 6.29 plot the overall accuracies and Tables 6.12 and 6.13 list the detailed results. As is observed with the single hidden layer case, much more of the larger network can be deleted with the performance still holding within a standard deviation of the error-rate estimate. For example, deleting roughly 85% of the 70-input network (4,730 weights left), still gives an overall accuracy over 98.2%. For the larger network, some improvement is also seen when a smaller percentage is deleted (13% deletion, 12,598 weights). This indicates that the OBD is having a beneficial effect, reducing the overfitting of the fully-connected network. In both networks, for deletion percentages under 10%, either the performance holds or improves over the fully-connected network. The procedure used to implement the algorithm is the same

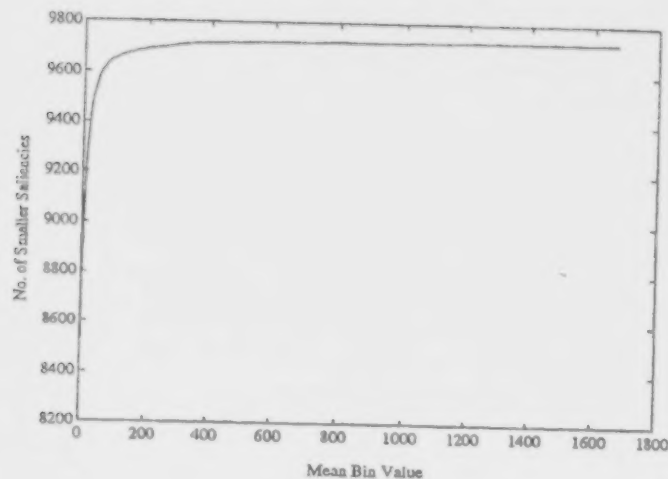


Figure 6.27: Cumulative Saliency Distribution Plot, 70 inputs, 130 hidden, 500 histogram bins used

as that for the single hidden layer case. Histograms are used to generate thresholds, and training is done for at least 1000 iterations after deletion. For both input sizes, and upto 50% deletion, the MSE after training is still under twice that given by the fully-connected networks.

W	AM	FM	ASK	QPSK	SSBU	SSBL	FSK1	FSK2	BPSK	CW	Overall
5450	95.2	96.1	96.3	99.5	99.5	99.6	98.3	100	98.1	100	98.26
5040	95.5	96.4	96.9	99.3	99.5	99.5	98.5	100	97.4	100	97.25
4771	94.6	96.1	96.6	99.3	99.5	99.5	98.3	100	97.7	100	98.16
4302	94.9	96.5	97.2	99.2	99.4	99.2	98.3	100	96.7	100	98.14
3875	95.1	95.9	97.3	99.2	99.4	99.3	98.2	100	96.1	100	98.05
3234	94.9	96.2	97.3	99.4	99.4	99.4	98.0	100	96.9	100	98.15
2124	92.5	92.4	96.3	97.4	98.9	98.6	96.5	99.9	95.7	100	96.82

Table 6.12: Sample OBD run for 42 inputs, 70 hidden 1, 30 hidden 2. W is the number of weights, and * indicates results of fully-connected network.

W	AM	FM	ASK	QPSK	SSBU	SSBL	FSK1	FSK2	BPSK	CW	Overall
14480	97.8	95.7	96.3	99.6	99.6	99.9	99.0	100	98.2	100	98.61
12598	97.4	96.0	97.3	99.6	99.6	99.8	99.1	100	97.5	100	98.63
11675	97.4	95.4	97.0	99.6	99.6	99.6	99.0	100	98.0	100	98.56
9951	97.4	95.3	96.7	99.5	99.5	99.7	99.2	100	97.7	100	98.50
4730	97.0	94.5	96.0	99.5	99.5	99.6	98.9	100	97.7	100	98.27
3693	0.1	93.8	96.6	99.6	99.4	99.7	99.2	100	96.3	100	88.47

Table 6.13: Sample OBD run for 70 inputs, 140 hidden 1, 30 hidden 2. W is the number of weights, and * indicates results of fully-connected network.

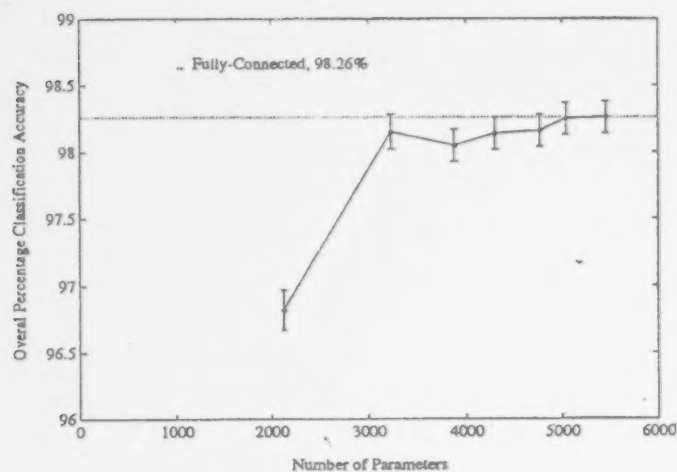


Figure 6.28: OBD for 42 inputs, 70 hidden 1, and 30 hidden 2. Fully connected network has 5,450 weights.

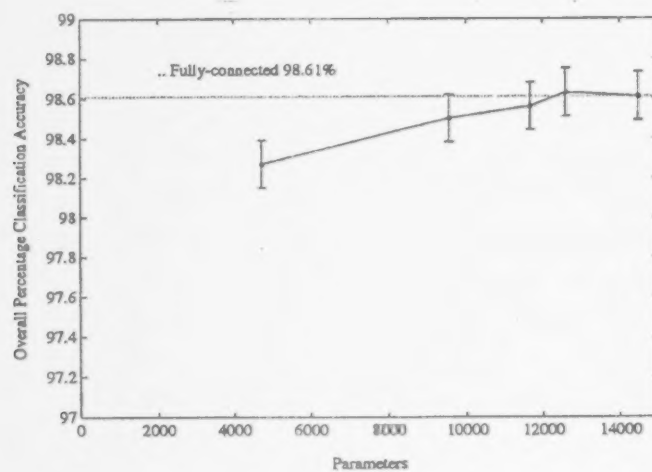


Figure 6.29: OBD for 70 inputs, 140 hidden 1, and 30 hidden 2. Fully connected network has 14,480 weights.

Chapter 7

Conclusions

7.1 Summary

In this research work, backpropagation neural network classifiers have been successfully implemented on the WARP systolic array processor for the automatic modulation recognition problem. The system handles 10 signal classes, a maximum bandwidth of 25kHz, and a wide SNR range (using additive white Gaussian noise). Three major signal preprocessing paradigms are chosen and tested for a difficult 2-class AM/FM trial problem to get an optimal preprocessor. All preprocessors perform well, over 80%, with best results occurring for the truncated, scaled-log Welch periodogram spectrum estimator. The regular periodogram with selective bin-averaging for dimensionality reduction also gives good performance, but the averaging being done in the Welch method gives less variability and hence slightly better results. Bispectrum preprocessing is found to generate excessively large feature set sizes which cannot achieve the same accuracy range as the periodograms due to the limited training set size.

Using the scaled-log Welch periodogram as the final preprocessor, both single and double hidden layer networks are found to give roughly the same performance on an independent test set. This indicates good class separability in the input domain. In addition, both 10kHz and 6kHz bandwidth feature sets show very close performance, hinting that the whole bandwidth is not required to get best results. Typically, for an

adequate network structure, individual class and overall accuracies can exceed those yielded by the conventional k-NN classifier, upper 90's percentages.

The optimal brain damage pruning algorithm provides additional optimization of the neural networks by giving significant weight reduction. Typically, deleting upto 10% of the weights on most networks does not yield an increase in the error rate on an independant test set. This means that the procedure is improving the generalization capabilities of the network by reducing the fitting of the training set. Furthermore, deleting upto 30-40% of the weights only leads to a slight increase in the error, indicating that the networks are robust. The training time for the neural network classifier is quite long, but once a network has been trained, classification can be done at almost real-time speed on the WARP array. Such speed is not possible with the k-NN.

7.2 Recommendations

For future work on the automatic recognition problem, some recommendations can be made:

- 1) Expand the range of the problem. This would mean more variability in the present signal classes (ie. data rates) and adding extra signal classes to the problem. Some important practical modulation formats which can be considered are minimum shift keying (MSK) and M-ary phase shift keying (M-PSK). As more and more classes are added, the preprocessing which is satisfactory for the 10-class problem may not be anymore. This would require the use of more elaborate preprocessing techniques such as bispectrum and cyclo-stationary spectrum to mention a few. As is found for bispectrum, these techniques require much larger training data sets, so the signal database has to be expanded significantly if such techniques are to be applied successfully.

- 2). The use of time-frequency analysis methods may prove worthwhile, especially if more signal classes are added. Some such methods include Wigner-Ville distributions, ambiguity functions, and chirplet transforms. Again, these methods require a increased dataset and the use of time-delay neural networks may be required.

3). Different neural networks, such as the radial basis function networks (RBF's), can be tried and their performance compared to that of the backpropagation network. These networks are quite powerful and typically yield results in the same range as the backpropagation networks. However, training data requirements for these networks can be significantly lower and hence this may be attractive for more advanced pre-processing techniques. In addition, another technique which breaks up the 10-class problem into smaller sub-problems can be tried. In such a method, rather than train a single network for all classes, smaller networks are trained for the sub-problems, and a "control" network used to selectively choose between the networks. Such an approach can give better results since less discriminatory ability is demanded from the feature set to solve the sub-problems. However, the training of such networks is more involved and time-consuming, and significant software modification has to be performed.

4). Finally, to model the problem even more accurately, real-world signals can be collected instead of using simulators. These signals would give much better figures on the real performance of a practical AMR system and would probably have significant non-Gaussian noise.

Appendix A

Overview of Modulation Schemes

A quick overview of the modulation classes is presented. In addition, the equations governing the digital implementations are also given. The software itself is written in C, and runs on *SUN Unix*. Interested readers are referred to the references [11] for very detailed descriptions of the software.

A.1 Amplitude Modulation

This format varies the amplitude of a sinusoidal carrier signal with respect to the voice signal to be transmitted. The general expression for the AM signal is [11]:

$$S(t) = A_c(1 + k_a m(t)) \cos(2\pi f_c t) \quad (\text{A.1})$$

where f_c is the carrier frequency (or IF), $m(t)$ is the voice signal to be transmitted, and k_a the modulation index. The bandwidth of the AM signal is equal to the highest frequency present in the message signal, limited to 6kHz by the voice filter. In addition, the signal is hardlimited to prevent carrier phase reversal (ie. $|k_a \max\{m(t)\}| \leq 1$). In its discrete form, for digital implementation, the in-phase component of the above signal can be written as:

$$I(nT_s) = A_c(1 + k_a m(nT_s))(\cos(2\pi f_c nT_s)) \quad (\text{A.2})$$

where T_s is the system sample period ($\frac{1}{f_s}$), and $m(nT_s)$ is the interpolated voice signal. It should be noted that the quadrature component, $Q(nT_s)$, contains no information

and only additive noise.

A.2 Frequency Modulation

In this scheme, the frequency of a sinusoidal carrier signal is varied as a function of the message signal:

$$S(t) = A_c \cos(2\pi f_c t + 2\pi k_f \int_0^t m(\tau) d\tau) \quad (\text{A.3})$$

where k_f is the frequency sensitivity (Hz/V). The bandwidth of FM signals is hard to specify in a closed form. Instead, the peak frequency deviation, Δf can be defined as $\Delta f = k_f A_{\max}$, where $A_{\max}$ is the maximum magnitude of the modulating signal. From this, the deviation ratio, D , is defined as the ratio of Δf to the maximum modulation frequency, W :

$$D = \frac{\Delta f}{W} \quad (\text{A.4})$$

Carson's rule uses the above quantities to estimate the bandwidth of the FM signal as:

$$B \approx 2\Delta f \left(1 + \frac{1}{D}\right) \quad (\text{A.5})$$

Therefore, by specifying the frequency deviation, Δf , the bandwidth of the signal can be controlled.

It can be shown [13] that the in-phase and quadrature components for a *baseband* FM signal are given by:

$$I(t) = A_c \cos(2\pi k_f \int_0^t m(\tau) d\tau) \quad (\text{A.6})$$

$$Q(t) = A_c \sin(2\pi k_f \int_0^t m(\tau) d\tau) \quad (\text{A.7})$$

For digital implementation, the above integral can be approximated by a sum, giving the discrete versions of Eqs. A.6 and A.7:

$$I(nT_s) = A_c \cos(2\pi k_f \sum_{i=1}^n m(iT_s)) \quad (\text{A.8})$$

$$Q(nT_s) = A_c \sin(2\pi k_f \sum_{i=1}^n m(iT_s)) \quad (\text{A.9})$$

Noise can be added to these baseband signals, and the results shifted up to the required IF.

A.3 Single Side Band Modulation

This modulation format reduces the bandwidth of the transmitted signal by only sending one sideband. This is due to the fact that for a real voice signal the positive and negative frequencies (baseband) contain the same information. The method of the Hilbert transform can be used to selectively eliminate a given sideband, where the Hilbert transform kernel is [6]:

$$h(t, \tau) = \frac{1}{\pi(\tau - t)} \quad (\text{A.10})$$

Consider a message signal, $m(t)$. Let $\hat{m}(t)$ be the Hilbert transform of $m(t)$,

$$\hat{m}(t) = m(t) * h(t) = \frac{1}{\pi} \int_{-\infty}^{+\infty} \frac{m(\tau)}{t - \tau} d\tau \quad (\text{A.11})$$

The properties of the Hilbert transform state that the Fourier transform of $\hat{m}(t)$ is now given by:

$$\mathcal{F}\{\hat{m}(t)\} = \hat{M}(f) = -jM(f)\text{sgn}(f) \quad (\text{A.12})$$

Now define the complex sum of a signal and its Hilbert transform, $z(t)$ as:

$$z(t) = m(t) + j\hat{m}(t) \quad (\text{A.13})$$

Therefore the Fourier transform of $z(t)$ now only has positive frequencies:

$$\mathcal{F}\{z(t)\} = Z(f) = \begin{cases} 2M(f), & f \geq 0 \\ 0, & f \leq 0 \end{cases} \quad (\text{A.14})$$

Similarly, by subtracting $\hat{m}(t)$ in Eq. A.13, the resultant sum will only have negative frequencies. Now, by modulating $z(t)$, half of the message signal's bandwidth is not transmitted. However, unlike the AM case, the signal being modulated is complex and therefore there is information in the quadrature channel. Stremler [6] defines the final modulated time signal as:

$$\phi_{SSB\pm}(t) = m(t)\cos(2\pi f_c t) \pm \hat{m}(t)\sin(2\pi f_c t) \quad (\text{A.15})$$

The discrete version of this technique computes the Hilbert transform of the up-sampled discrete message signal $m(nT_s)$. This is done by convoluting the message signal with the discrete Hilbert impulse response, given by [11]:

$$h(n) = \begin{cases} \frac{2}{\pi} \frac{\sin^2(\frac{\pi n}{2})}{n}, & n \neq 0 \\ 0, & n = 0 \end{cases} \quad (\text{A.16})$$

The sum/difference of $m(nT_s)$ and its Hilbert transform is then modulated to a carrier frequency, and the discrete equations for the inphase and quadrature channel can be derived from Eqs. A.2 and A.15:

$$I(nT_s) = (1 + k_a m(nT_s)) \cos(2\pi f_c nT_s) \quad (\text{A.17})$$

$$Q(nT_s) = k_a \hat{m}(nT_s) \sin(2\pi f_c nT_s) \quad (\text{A.18})$$

For suppressed carrier transmission, the modulation index is given by:

$$k_a = \frac{1}{\sqrt{C_f}} \quad (\text{A.19})$$

where C_f is defined as the carrier feedthrough power. Therefore, when the carrier feedthrough is low, the message component is relatively stronger than the carrier component, and vice versa.

A.4 Amplitude Shift Keying

This transmission scheme modulates a sinusoidal carrier to send binary (digital) data. Specifically, a signal is transmitted to send a 1 and no signal is transmitted for a 0. In a coherent system, one can set the carrier to be a multiple of half the bit rate, ensuring that the carrier is 0 at each bit transition. However, since the exact carrier frequency of an intercepted signal is assumed unknown, non-coherent (OOK, On-Off-Keying) signals are used. The mathematical form of this modulation type is:

$$S(t) = \sum_{n=-\infty}^{\infty} A_c b_n(t) \cos(2\pi f_c t) \quad (\text{A.20})$$

where $b_n(t)$ represents the instantaneous value of the n -th bit of the bitstream.

$$b_n(t) = \begin{cases} 1, & \text{if 1 sent at } nT_s \leq t \leq (n+1)T_s, \\ 0, & \text{if 0 sent at } nT_s \leq t \leq (n+1)T_s, \\ & \text{or if } t < nT_s \text{ or } t > (n+1)T_s, \end{cases} \quad (\text{A.21})$$

The discrete implementation is straightforward, since the the interpolated bistream ($b_n(iT_s)$), can be directly applied to the transmit filter, noise added, and the result shifted up to f_c . Note that as in the AM case, there is no information carried in the quadrature channel. Specifications state data rates for ASK ranging from 30 to 100 bauds. Compared to the other digital modulations, these rates are very low and result in a narrower bandwidth.

A.5 Binary Phase Shift Keying

In this scheme, the carrier phase is changed to send the two bit values: 0 degrees for 1, 180 degrees for 0. As a result, this modulation scheme is very similar to ASK. Instead of sending no signal for 0, a negative signal level is sent (ie. 180 degree phase shift). The mathematical form is identical to Eq. A.21 except that now $b_n(t)$ is:

$$b_n(t) = \begin{cases} 1, & \text{if 1 sent at } nT_s \leq t \leq (n+1)T_s, \\ -1, & \text{if 0 sent at } nT_s \leq t \leq (n+1)T_s, \\ & \text{or if } t \leq nT_s \text{ or } t \geq (n+1)T_s, \end{cases} \quad (\text{A.22})$$

Specifications state that BPSK signals can only have the one of the following bit rates: 75, 150, 300, 600, or 1200.

A.6 Quadrature Phase Shift Keying

In this scheme, the carrier phase takes on one of four values, depending upon the *dibit* (bit pair) being sent:

Dibit, d_i	Bit Pair, b_1b_2	Carrier Phase, ϕ_i
d_1	00	$\frac{\pi}{4}$
d_2	01	$\frac{3\pi}{4}$
d_3	11	$\frac{5\pi}{4}$
d_4	10	$\frac{7\pi}{4}$

Hence the mathematical form of this scheme is:

$$S(t) = A_c \cos(2\pi f_c t + \theta_{d_n}(t)) \quad (\text{A.23})$$

where $\theta_{d_n}(t)$ is the phase corresponding to the n -th dibit being sent at time t :

$$\theta_n(t) = \begin{cases} \phi_i, & \text{if dibit } d_i \text{ being sent at time } nT_s \leq t \leq (n+1)T_s \\ 0, & \text{otherwise} \end{cases} \quad (\text{A.24})$$

It can be show mathematically [1] that the above scheme can be implemented by using two BPSK signals with quadrature carriers (90 degrees out of phase), one for each bit in the dibit pair. The even bits modulate one carrier and the odd bits the other, with the signals being summed before the transmit filter. Hence the digital implementation is very similar to the BPSK case, with the two upsampled bitstreams being summed, filtered, and noise added. However, unlike the BPSK case, there is information being carried in the quadrature channel.

A.7 Frequency Shift Keying

Unlike the previous binary modulation schemes where either the amplitude or phase is modified to send binary values, this scheme transmits two different frequencies (f_0, f_1) for the bit values. These two transmission frequencies are related by the h parameter:

$$h = T_b(f_0 - f_1) \quad (\text{A.25})$$

where T_b is the bit duration. Choosing the carrier frequency, f_c as the midpoint of the above frequencies (ie. $f_c = \frac{1}{2}(f_0 + f_1)$), the scheme can be written mathematically

as:

$$S(t) = A_c \cos(2\pi f_c t + \theta(t)) \quad (\text{A.26})$$

where θ_0 is phase at beginning of bit period and $\theta(t)$ is:

$$\theta(t) = \pm\theta_0 + \frac{\pi h}{T_b} t \quad (\text{A.27})$$

where θ_0 is the phase at the beginning of the bit period, $\frac{1}{T_b}$ and $+$ sign for 1 and $-$ for 0. Using trigonometric identities, Eq. A.26 can be rewritten as

$$S(t) = \cos(\theta(t)) \cos(2\pi f_c t) - \sin(\theta(t)) \sin(2\pi f_c t) \quad (\text{A.28})$$

Using this new form, the equations for the digital implementation of the in-phase and quadrature channels become:

$$I(n) = \cos\left(\frac{\pi h}{T_b} \sum_{i=1}^n b_n(iT_s)\right) \quad (\text{A.29})$$

$$Q(n) = \sin\left(\frac{\pi h}{T_b} \sum_{i=1}^n b_n(iT_s)\right) \quad (\text{A.30})$$

where $b_n(iT_s)$ is the upsampled bistream (ie. input bistream upsampled from $\frac{1}{T_b}$, transmitter bit frequency, to $\frac{1}{T_s}$, system sampling frequency) Note that the summation in the above expressions helps generate θ_0 , the initial phase. The above baseband signal is filtered and shifted to the required frequency, and then noise added to generate the required IF signal. Specifications require two FSK signal classes: slow data rates (45-100 bauds) and fast data rates (16-19.2kbps). As a result, the transmit filter bandwidth is specified as 6kHz and 25kHz respectively.

A.8 Continuous Wave Modulation

In this modulation, there is no message signal being sent. There is only a carrier tone, at a fixed frequency, f_c :

$$S(t) = A_c \cos(2\pi f_c t) \quad (\text{A.31})$$

Therefore the equations and implementation for this class are identical to those for AM, with zero message signal and/or modulation index.

Appendix B

Signal Specifications

The following tables contain specific details of the signal types which have been generated in the database. Most of these are taken from *Attachment B* in the *Statement of Work* [5], which states the requirements for various signal parameters. Any major choices made regarding parameters given in the specifications are justified accordingly. Please note that the SNR for all signal types is selected from a Gaussian distribution, mean 11dB, standard deviation 6dB, minimum 5dB and maximum 25dB.

B.1 Amplitude Modulation

<i>AM Characteristic</i>	<i>Value</i>
Mean IF	15kHz
IF standard deviation	600 Hz
Minimum IF	13.5kHz (-2.5 standard deviations)
Maximum IF	16.5kHz (+2.5 standard deviations)
Filter Bandwidth at Transmitter (Filter1)	6kHz
Modulation Index mean	70%
Modulation Index Std. Dev.	15%
Modulation index minimum	50% (-1.33 standard deviations)
Modulation index maximum	90% (+1.33 standard deviations)

Since the AM signals are modulated by speech waveforms, and it is assumed that speech signals are bandlimited to 3kHz, a bandwidth of 6kHz results. 10% of this value is 600 Hz, the standard deviation. The modulation index is chosen to have a Gaussian distribution with a mean halfway between the values specified [5].

B.2 Frequency Modulation

<i>FM Characteristic</i>	<i>Value</i>
Mean IF	15kHz
IF standard deviation	750 Hz
Minimum IF	13kHz (-2.67 standard deviations)
Maximum IF	17kHz (+2.67 standard deviations)
Filter Bandwidth at Transmitter (Filter1)	25kHz
Frequency Deviation mean	3.75kHz
Frequency Deviation Std. Dev.	900 Hz
Frequency Deviation minimum	1.5kHz (-2.5 standard deviations)
Frequency Deviation maximum	6.0kHz (+2.5 standard deviations)

Most of the above values are directly derived from the specifications. The frequency deviation is again chosen to have a Gaussian nature, centered between the maximum and minimum values specified in *Attachment B* [5]. By appropriately adjusting the frequency sensitivity, the bandwidth of the resultant signal can be approximately made large ($\approx 25\text{kHz}$) for large frequency deviations and small for smaller deviations (using Carson's Rule, Eq. A.5).

B.3 Single Side Band Modulation

<i>SSB Characteristic</i>	<i>Value</i>
Mean IF	15kHz
IF standard deviation	270 Hz
Minimum IF	14.25kHz (-2.78 standard deviations)
Maximum IF	15.75kHz (+2.78 standard deviations)
Filter Bandwidth at Transmitter (Filter1)	2.7kHz
Carrier Feedthrough Power Mean:	0.008
Carrier Feedthrough Power Std. Dev.	0.014
Carrier Feedthrough Minimum:	0.005 (-0.21 standard deviations)
Carrier Feedthrough Maximum:	0.05 (+3.0 standard deviations)

The lower cutoff for the voice filter (Filter1) is 300 Hz and the upper cutoff is 3 kHz. Since the signal has attenuated carrier, the bandwidth is roughly 2.7kHz. 10% of this value is 270 Hz, the chosen IF standard deviation.

The carrier feedthrough power parameters are all fractions of the carrier power itself. These values give an average attenuation of 42dB, with maximum and minimum values of 46dB and 26dB respectively. These values are close to the 25dB to 40dB range specified in *Attachment B* [5]. Since the mean is very close to the high attenuation end, the signals have very little carrier power.

B.4 Amplitude Shift Keying

<i>ASK Characteristic</i>	<i>Value</i>
Mean IF	15kHz
IF standard deviation	13 Hz
Minimum IF	14970 Hz (-2.31 standard deviations)
Maximum IF	15030 Hz (+2.31 standard deviations)
Filter Bandwidth at Transmitter (Filter1)	3kHz
Data Rate Distribution	30 Hz to 100 Hz, uniform

The mean data rate is 70 Hz. This gives a bandwidth of 130 Hz (mean) and 10% of this value is used as the standard deviation for the IF estimate.

B.5 Quadrature Phase Shift Keying

<i>QPSK Characteristic</i>	<i>Value</i>
Mean IF	15kHz
IF standard deviation	500 Hz
Minimum IF	13.8kHz (-2.4 standard deviations)
Maximum IF	16.2kHz (+2.4 standard deviations)
Data Rates	2400 bps, 4800 bps, 9600 bps
Filter Bandwidth at Transmitter (Filter1)	2.5 x data rate

It should be noted that the QPSK signals can only occur at three different modem baud rates. These values are selected from a uniform distribution. The mean baud rate is therefore 4800 bps. The width of the main lobe in a QPSK spectrum is equal to the baud rate, and if this width is used as an approximation to the bandwidth, a value of 4.8kHz results. 10% of this value is 480 Hz and it is rounded upwards to 500 Hz and set equal to the IF standard deviation.

B.6 Binary Phase Shift Keying

<i>BPSK Characteristic</i>	<i>Value</i>
Mean IF	15kHz
IF standard deviation	60 Hz
Minimum IF	14850 Hz (-2.5 standard deviations)
Maximum IF	15150 Hz (+2.5 standard deviations)
Data Rates	75 bps, 150 bps, 300 bps, 600 bps, 1200 bps
Filter Bandwidth at Transmitter (Filter1)	3.0 x data rate

It should be noted that the BPSK signals can only occur at five different modem baud rates. These values are selected from a uniform distribution. The mean baud rate is therefore 300 bps. The width of the main lobe in a BPSK spectrum is equal to the twice the baud rate, and if this width is used as an approximation to the bandwidth, a value of 600 Hz results. 10% of this value is 60 Hz, the chosen standard deviation. Note that since this parameter is not so large (relatively) for higher modem baud rates, the IF estimation error will be less (relatively) for these rates.

B.7 Frequency Shift Keying

<i>Slow-Rate FSK (FSK1) Characteristic</i>	<i>Value</i>
Mean IF	15kHz
IF standard deviation	220 Hz
Minimum IF	14.5kHz (-2.27 standard deviations)
Maximum IF	15.5kHz (+2.27 standard deviations)
Filter Bandwidth at Transmitter (Filter1)	- 6kHz
Mean Data Rate	72 bps
Data Rate Std. Dev.	12 bps
Data Rate Minimum	45 (-2.25 standard deviations)
Data Rate Maximum	100 (+2.33 standard deviations)
h distribution	2.8 to 28, uniform

The h parameter defines the frequency separation (h times bit rate). The bit rate is chosen in a Gaussian manner, in the range 45 bps to 100 bps (*Attachment B*) [5]. The h values however are chosen in a uniform manner. Using h and the bit rate, the transmission frequencies can be computed Eq. A.25. Thus, using the mean values, the average frequency separation and bandwidth can be computed. The mean h value is 15.4 and when multiplied by an average data rate of 72 bps, gives a frequency separation of 1108.8 Hz. The bandwidth is about two times the frequency separation for FSK, roughly 2.2kHz. 10% of this value is 220 Hz, the IF standard deviation. The maximum possible bandwidth is roughly the product of two times the maximum h value and maximum data rate, 5.6 kHz. Thus a transmitter filter bandwidth of 6kHz is chosen to pass all spectral components of the bandwidth.

<i>Fast-Rate FSK (FSK2) Characteristic</i>	<i>Value</i>
Mean IF	15kHz
IF standard deviation	840 Hz
Minimum IF	12.84kHz (-2.70 standard deviations)
Maximum IF	17.16Hz (+2.70 standard deviations)
Filter Bandwidth at Transmitter (Filter1)	25kHz
Mean Data Rate	17.6 kbps
Data Rate Std. Dev.	600 bps
Data Rate Minimum	16 kbps (-2.67 standard deviations)
Data Rate Maximum	19.2 kbps (+2.67 standard deviations)
h distribution	0.50 to 0.65 uniform

The h parameter defines the frequency separation (h times bit rate). The bit rate is chosen in a Gaussian manner, in the range 16 kbps to 19.2 kbps (*Attachment B*). The h values however are chosen in a uniform manner. Thus, using the mean values, the average frequency separation and bandwidth can be computed. The mean h value is 0.575 and when multiplied by an average data rate of 17.6 kbps, gives a frequency separation of 10.12 kHz. Most of the signals bandwidth lies below this range, so an IF standard deviation of 840 Hz is chosen. The maximum possible bandwidth possible is precisely the product of two times the maximum h value and maximum data rate, 24.96 kHz. Thus a transmitter filter bandwidth of 25kHz is chosen to pass all spectral components of the bandwidth.

B.8 Continuous Wave Modulation

<i>CW Characteristic</i>	<i>Value</i>
Mean IF	15kHz
IF variance	1000 Hz
Minimum IF	13.0kHz (-3.0 standard deviations)
Maximum IF	18.0kHz (+3.0 standard deviations)

Most F's for the other modulation types lie within about 2kHz of the mean IF. To cover this range adequately, a standard deviation of 1kHz is specified for this "no-information" class.

Bibliography

- [1] Haykin, S., *Digital Communications*, John Wiley & Sons, 1989.
- [2] Fukunaga, K., *Introduction to Statistical Pattern Recognition*, Academic Press, London, 1972.
- [3] Hand, D., *Discrimination and Classification*, John Wiley & Sons Ltd., New York, 1981.
- [4] Haykin, S., *Introduction to Analog and Digital Communications*, John Wiley & Sons, New York, 1989.
- [5] Lamontagne, R., "Automatic Modulation Recognition Based on Spatial Pattern Classification Using Artificial Neural Networks", *Statement of Work for DSS Contract #027ST.W7714-1-9531*, Defense Research Establishment Ottawa, 1991.
- [6] Stremmler, F., *Introduction to Communication Systems, Second Edition*, Addison-Wesley Publishing Company, Massachusetts, 1982.
- [7] Lapin, L., *Statistics: Meaning and Method*, Harcourt Brace Jovanovich, Inc., New York, 1975.
- [8] Beyer, W.H. (Editor), *CRC Standard Mathematical Tables, 28th Edition*, CRC Press, Inc., Boca Raton, Florida, 1987.
- [9] Rabiner, L. and Gold, B., *Theory and Application of Digital Signal Processing*, Prentice-Hall Inc., 1975.

- [10] Duda, R. and Hart, P., *Pattern Classification and Scene Analysis*, John Wiley & Sons, Toronto, 1973.
- [11] Lazaris-Brunner, K., "Modulation Simulation Program: Operator's Manual/ Programmer's Manual, Version 1.00", Calian Communications Systems Ltd., CCS File DREO-16, October 1990.
- [12] Oppenheim, A. and Schafer, R., *Digital Signal Processing*, Prentice Hall, New Jersey, 1975.
- [13] Dixon, R., *Spread Spectrum Systems*, John Wiley & Sons, 1976.
- [14] Welch, P.D., "The Use of Fast Fourier Transforms for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms", *IEEE Transactions on Audio and Electroacoustics*, Vol. AU-15, No. 2, June 1967, pp.70-73.
- [15] Nikias, C. and Raghuveer, M., "Bispectrum Estimation: A Digital Signal Processing Framework", *Proceedings of the Workshop on Higher-Order Spectral Analysis*, 1987, pp.869-891.
- [16] Nikias, C. and Raghuveer, M., "Bispectrum Estimation: A Parametric Approach", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, VOL. ASSP-33, No. 4, October 1985, pp.1213-1229.
- [17] Rao, S. and Gabr, M., *Lecture Notes in Statistics: An Introduction to Bispectral Analysis and Bilinear Time Series Models*, Springer-Verlag, New York, 1984.
- [18] Devijver, P. and Kittler, J., *Pattern Recognition: A Statistical Approach*, Prentice-Hall, 1982.
- [19] Haykin, S., *Introduction to Neural Networks*, MacMillan, (to be published), 1992.
- [20] Gish, H., "A Probabilistic Approach to the Understanding and Training of Neural Network Classifiers", *IEEE*, 1990, pp. 1361-1364.

- [21] Hampshire II, J., Pearlmutter, B., "Equivalence Proofs for Multi-Layer Perceptron Classifiers and the Bayesian Discriminant Function", *Connectionist Models: Proceedings of the 1990 Summer School*, Morgan Kaufmann Publishers, 1991, pp. 159-172.
- [22] Flannery, B. and Press, W. and Teukolsky, S. and Vetterling, W., *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, New York, 1988.
- [23] Baum, E. and Haussler, D., "What Size Net Gives Valid Generalization?", *Neural Information Processing Systems 1*, Morgan Kaufmann Publishers Inc., 1989, pp.81-90.
- [24] Ahmad, S. and Tesauro, G., "Scaling and Generalization in Neural Networks: A Case Study", *Neural Information Processing Systems 1*, Morgan Kaufmann Publishers Inc., 1989, pp. 160-168.
- [25] Muller, B. and Reinhardt, J., *Neural Networks: An Introduction*, Springer-Verlag, New York, 1990.
- [26] Le Cun, Y., Denker J. and Solla, S., "Optimal Brain Damage", *Neural Information Processing Systems 2*, Morgan Kaufmann Publishers Inc., 1990, pp.598-605.
- [27] Tseng, P., *A Systolic Array Parallelizing Compiler*, Kluwer Academic Publishers, Boston, 1990.
- [28] Lam, M., *A Systolic Array Optimizing Compiler*, Kluwer Academic Publishers, Boston, 1989.
- [29] *Warp User Manual Version 3.0*, General Electric Company, September 1988.

Part II

**Automatic Modulation
Recognition - Software User's
Manual**



1 Introduction

This part of the document contains details on *C* programs written for the DREO AMR contract. The purpose and functionality of each program is described. The programs include preprocessing programs (PSD averaging, modified Welch periodogram, bispectrum), file generating programs, and classifier programs.

2 Preprocessing Software

Detailed descriptions of all the preprocessors can be found in the Final Report for the contract. Only software details are presented in this manual. All programs assume that the input files are in ASCII format, with successive samples separated by either spaces or return characters.

2.1 Power Spectral Density Averaging, PSD.C

This program takes an input signal of a specified length, and computes the PSD averaged output. The length of the signal, the FFT size, the bandwidth chosen, and the number of filters are all to be specified by the user in the constants declaration at the start of the code.

For example, the research involves signals of length 8,250 samples. Hence an FFT size (`SAMPLE_SIZE` constant) is set to 8192. The bin corresponding to the centre IF of 15kHz, `MID_POINT` constant, is given by:

$$\text{MID\_POINT} = \frac{f_s(IF)}{\text{SAMPLE\_SIZE}} \quad (1)$$

where IF is the IF frequency in Hz, and f_s is the sampling rate. Hence for a 15kHz IF and 75kHz sampling and 8192 FFT size, the `MID_POINT` is roughly 1638. In other words, bin 1638 in the power spectrum representation corresponds to 15kHz. This is because the magnitude of the FFT from bins 0 to $\frac{\text{SAMPLE\_SIZE}}{2}$ corresponds to frequencies 0 to $\frac{f_s}{2}$ Hz.

To reduce the input size to the classifiers, a filter set is specified in an input file which specifies the number of filters and each filter's range in bin values. This allows the user to control how much bandwidth is covered and also the overlap between successive filters. For example, consider an 10 filter

set, the file for it (FILTER_10) is given by:

```
10
1203 1314
1314 1395
1395 1476
1476 1557
1557 1638
1638 1719
1719 1800
1800 1881
1881 1962
1962 2073
```

The first line of this file specifies the number of filters, 10, and the subsequent lines specify the range of each filter. For example, the first filter, corresponding to the lower frequency range has bin range from 1203 to 1314. This corresponds to a frequency range of roughly 11.01kHz to 12.03kHz, bandwidth roughly 1020Hz. The middle filters are a bit narrower, for frequencies close to the IF. This allows the user to specify any filter setup, but a filter file has to be generated beforehand and manually entered, and for large filter sets can be very tedious. The user must make sure that the number of filters being specified is less than or equal to the MAX_FILTERS constant in the code. Filter files for 10, 20, 30, 60, and 80 filters have been generated and are included in the software.

Hence the program computes the PSD-averaged energy in each filter and outputs the results to a single line in an output file, with the outputs being separated by a space. The exact format of the output is:

`filter1_output, filter2_output, filter3_output,...,filterN_output`

The code for this program is fully commented, and the runline for the program is:

`psd input_file output_file filter_file`

where *input_file* is the input file (with more than SAMPLE_SIZE points), *output_file* is the file containing the filter outputs, and *filter_file* is the filter set description file.

2.2 Welch's Modified Periodogram, PER.C

This program takes an input signal of a specified length, and computes the modified Welch periodogram of it. The length of the signal, the FFT size, the number of frames averaged, frame overlap, and bandwidth processed are all to be specified by the user in the constants declaration at the start of the code.

The total number of points to be processed is given by the SAMPLES constant. The FFT size (frame length) is given by SAMPLE_SIZE, and OVERLAP and SEPARATION define the overlap between successive frames. They are related by

$$\text{SAMPLE\_SIZE} = \text{OVERLAP} + \text{SEPARATION} \quad (2)$$

and the total number of frames processed is given by:

$$K = \frac{\text{SAMPLES}}{\text{SEPARATION}} \quad (3)$$

The FFT size controls the spectral resolution of the technique, and by specifying it along with the SAMPLE_SIZE, the number of averages taken can be computed. Increasing FFT sizes gives better resolution, but there is an increase in the variability of the estimates. Reducing FFT sizes, on the other hand reduces variability (since more frames being averaged), but decreases resolution (less bins covering same bandwidth).

Typically, only a certain bandwidth is to be extracted in the estimate of the periodogram. Therefore the bin values corresponding to the lower and upper bounds of this bandwidth are also user specified, LOWER and HIGHER constants. These values are computed as:

$$\text{LOWER} = \frac{(\text{SAMPLE\_SIZE})f_l}{f_s} \quad (4)$$

and

$$\text{HIGHER} = \frac{(\text{SAMPLE\_SIZE})f_h}{f_s} \quad (5)$$

SAMPLE_SIZE	Frames Averaged	LOWER	HIGHER	Total Outputs
128	64	17	34	18
256	32	35	69	35
512*	16*	68*	137*	70*
1024	8	138	272	135

Table 1: Summary of PER.C constant definitions used in research, SAMPLES=8192, Zero Overlap. * denotes values chosen for final preprocessor.

where f_s is the sampling frequency in Hz, and f_l and f_h are the lower and upper bounds of the bandwidth to be extracted, respectively. For example, for 8,192 sample points (SAMPLES), FFT size 512 (SAMPLE_SIZE), zero overlap (OVERLAP) and a sampling rate of 75kHz (f_s), a 10kHz bandwidth is given by bin values 68 and 137 (LOWER and HIGHER). Table 0.1 table summarizes values used for 10kHz bandwidth in the research.

The program sums the magnitudes of the values of each bin in the desired range and outputs the results to a single-line output file, the entries separated by spaces. The total number of entries written out to the line is (HIGHER-LOWER+1).

The code for the program is fully documented and all major functions explained. The program basically goes through the samples, grabbing frames, computing the periodogram estimates, and updating a running sum. To implement overlapped frames, two data arrays are declared with the frames being read into alternating arrays. This procedure is fully described in the commenting, and is done to speed up the program. The runline for the program is:

per *input_file* *output_file*

where *input_file* is the input file (with more than SAMPLE_SIZE points), and *output_file* is the file containing the filter outputs.

2.3 Bispectrum, BSP.C

This program takes an input signal of a specified length, and computes its optimally-windowed bispectrum estimate. The length of the signal, the FFT size, and overlap between successive frames are all to be specified by the user in the constants declaration at the start of the code.

The output of the program is a discrete sampling of the *magnitude* of the first bispectral slice. To get the complex bispectrum or phase, minor modifications can be made to the code to output them (indicated in comments). However, this will also double the output dimensionality. The program takes SAMPLES points and uses SAMPLE_SIZE FFT's to compute bispectral estimates using the method specified by Nikias (see Final Report). This method basically computes FFT's of the frames, and then computes products at certain frequencies to generate estimates in the bispectral domain. Again, as is possible in the Welch periodogram estimator program, the FFT frames can be overlapped by specifying a non-zero OVERLAP constant, and adjusting the SEPARATION constant appropriately. The program allows no averaging of bins in the bispectral domain to reduce input dimensionality, and therefore all points in the first slice are written to a file. The number of points written out is governed by the FFT size, SAMPLE_SIZE:

$$\text{Number of output points} = \sum_{i=0}^{\text{SAMPLE\_SIZE}} \left(\frac{\text{SAMPLE\_SIZE}}{2} - 2i \right) \quad (6)$$

The bispectral outputs are written in a single line output with successive entries separated by a space. The commented code fully describes the functionality of the program and all variables used. The program basically goes through all the samples, grabbing frames, computing bispectral slices for each and updating a running complex sum. At the end, the magnitude of complex sum is taken and optimally windowed to generate the output. The runline for this program is:

bsp *input_file* *output_file*

where *input_file* is the input file (with more than SAMPLE_SIZE points), and *output_file* is the file containing the filter outputs.

3 File Generating Software

The previous programs all generate single file outputs corresponding to a single input signal from a certain class. However, for training and testing sets, data from many different input signals from all classes needs to be combined. Further, for backpropagation network training purposes, the data should even be randomly rearranged among the classes. The going from a large database of unprocessed files for each modulation class to a single continuous preprocessed file for this database is big issue, and involves two major steps.

In the first major step, each class is processed separately. Typically, the files for a given class have names S1.ASC, S2.ASC, S3.ASC,... Each of these files needs to be preprocessed, and the output written to single-line output files, with the *same* corresponding number. For example, the preprocessed output for file S1.ASC (first signal) can be stored in file S1.Pxxx, and for S2.ASC, S2.Pxxx, so on and so forth, where xxx is half the FFT size. One can write a jobfile using the UNIX *chmod u+x* command to run the job in the background and generate the files. A sample jobfile for PSD-averaging is:

```
psd ....S1.ASC temp/S1.P256
psd ....S2.ASC temp/S2.P256
psd ....S3.ASC temp/S3.P256
```

..
..

where the specifies the pathname to the signal database directory of the pertinent class and temp is a temporary directory to hold intermediate preprocessing results. Once all these single-line files have been preprocessed and the S*.Pxxx files stored in temp, another program is run to combine these files in numerical order (ie. S1.Pxxx, S2.Pxxx, S3.Pxxx). The files are combined in numerical order by the COMBINE.C program. The UNIX *cat* command could be used here, but incase some of the preprocessed files are bad (ie. NaN or -Inf in output), the numerical ordering can be used to track down the bad ones.

COMBINE.C combines a prespecified number (COMBINE constant) of single-line preprocessed files into a larger file containing that many lines. The suffix ".Pxxx" of the files are specified by the SUFFIX string constant

and the number of entries per line (ie. feature vector length) is given by the VECTOR constant. The order in which the files are combined is given in the NUMBERS file. This file has to be present for the program to run. For example, if the format of the NUMBERS file is:

```
1
2
3
.
.
10
```

then 10 preprocessed files will be combined in order S1.Pxxx, S2.Pxxx, ..., S10.Pxxx. The first line of the output file will contain the line from S1.Pxxx, the second from S2.Pxxx, and so on and so forth. Note that by specifying the order in a NUMBERS file, one can specify any order of the preprocessed files to be combined in. This allows one to omit bad files and also avoid coding problems assuming a continuous numerical range.

The code for this program is very straightforward, and is fully commented, explaining each step. Basically, the program loops through all entries in the NUMBERS file, generates the appropriate file name, reads in the file and copies it to the output file. The runline is:

```
combine input_directory output_file
```

where *input_directory* is the directory containing the single-line preprocessed files, and *output_file* is the combined output file, COMBINE lines long, each line with VECTOR entries. Please note that the NUMBERS file has to be resident on the same directory as the program itself and must contain at least COMBINE entries.

The second major step in the file generation is regarding randomly rearranging the training vectors (lines) to prevent biasing in backpropagation training. Typically, for testing, combining files is not a problem. For example, if preprocessed files of a certain length have been generated for each class (using COMBINE.C), they can be individually tested and simple programs written (ie. in MATLAB) to count errors. Otherwise, these individual files

can be concatenated in a certain order using the UNIX *cat* command, and then this larger file tested in one go. For the optimal preprocessor, log Welch periodogram with FFT size 512 and 70 and 42 bin output, the testing and validation files are generated in this manner. For example, the testing files are LOG_div_10.TEST and SHORT_div_10.TEST for 70 and 42 bin input respectively. The structure of both of these files is the following:

- 1000 AM vectors
- 1000 FM vectors
- 1000 ASK vectors
- 1000 QPSK vectors
- 1000 SSBV vectors
- 1000 SSBL vectors
- 1000 FSK1 vectors
- 1000 FSK2 vectors
- 1000 BPSK vectors
- 1000 CW vectors

For the validation sets, the files are LOG_div_10.VALID and SHORT_div_10.VALID, large and small input respectively. In these files, the ordering of the classes is the same as in the .TEST files, except that now there are only 100 vectors per class. The individual files for each class which are concatenated to generate the above files, containing the testing and validation vectors, are also stored and the names are:

*<modulation_format>*P256.*<type>*

where *<modulation_format>* is either AM, FM, ASK, QPSK, SSBV, SSBL, FSK1, FSK2, BPSK, or CW, and *<type>* is either TEST or VALID. For example, the AM test vectors are stored in the file AMP256.TEST.

To generate randomly combined files for the backpropagation training, however, another program has to be run. As described above, COMBINE.C is used to generate files for each class, and each file must contain the same number of lines (ie. feature vectors, unbiased). Once these files have been generated, the RANDCOMBINE.C program is used to randomly mix up the feature vectors (lines) from these files into a larger training file, and to generate a desired output file. The lines of the desired output file basically

state which classes the corresponding lines of the random file belong to (ie. it labels the training vectors).

The RANDCOMBINE.C program requires constants to be specified by the user. First of all, the number of classes (NUM_CLASS constant) and number (NCLASS constant) and size (VECTOR constant) of feature vectors from each class being combined have to be specified. This means that the previous runs with the COMBINE.C program should have produced NCLASS files, one for each class, each of length NCLASS lines. In addition, for the desired output files, the user has to specify the constants LTHRESH and UTHRESH, lower and upper thresholds, respectively. Typically, LTHRESH is a small number, very close to, but not equal to zero, and UTHRESH is close to, but not equal to, unity (i.e $1 - LTHRESH$). This prevents saturation later on in backpropagation training. Hence, the line written to the desired output file for a line corresponding to the n -th class is a sequence of n floating point numbers, with the n -th number being equal to UTHRESH and all the rest LTHRESH.

The program runs by simply choosing a class from a random uniform distribution, and then reading a line from the file corresponding to the particular class and copying it to the random output file. This process is repeated until all lines from all class files are processed. The number of lines read from each class is recorded so that when one class runs out before another, it is no longer considered. The code for this program is fully documented and easy to follow. The runline for RANDCOMBINE.C is:

```
randcombine class1_file class2_file... class<n>_file random_file  
desired_output_file
```

where *class*_file* are the preprocessed files for the classes to be combined (each of NCLASS lines or more) and *random_file* and *desired_output_file* are the randomly ordered and label files respectively.

The training file for the Welch preprocessor, FFT size 512, and 42 and 70 input bins already exist: LOG_div_10.TRAIN and SHORT_div_10.TRAIN. The label file for *both* of these files is called TRAIN.OUT. These files each contain 1,800 training vectors from each of the 10 classes, and hence have an overall length of 18,000 lines. Note that the individual files from which these randomly-ordered training files are generated are also available, and the names are:

`<modulation_format>P256.TRAIN`

where `<modulation_format>` is either AM, FM, ASK, QPSK, SSB, SSBL, FSK1, FSK2, BPSK, or CW. Each of these files is 1,800 lines long.

4 Classifier Software

There are two major types of classifiers used in the research. The first is the standard k-NN and the other is the backpropagation neural network. The code for the backpropagation networks is adapted from WARP W2 code and is to run on a *SUN Unix*. All classifiers used preprocessed files outputted from the preprocessing software.

4.1 k-NN Classifier, KNN.C

This program takes a labelled input dataset and a test file of vectors and uses the k-NN method to classify the vectors. The input files consist of a randomly arranged vector file and corresponding label file, specified in the exact same manner as the outputs of the RANDCOMBINE.C program.

Specifically, the program reads in the input files, which should contain an equal number of examples (TRAIN) from each of the classes (CLASS). The number of vectors to be tested is specified by the TEST constant. The dimensionality of the input vector is given by the VECTOR constant, and K is the value of k . The program first loops through the input file and loads up the input data into an array corresponding to the class. Next, test vectors are read in and compared against *all* training vectors and the closest class is outputted.

The code of the program is fully commented and very straightforward. The runline for the program is:

```
Knn input_file label_file test_file output_file1 output_file2
```

where *input_file* is the randomly rearranged vector file, and *label_file* is the corresponding label file, outputs of RANDCOMBINE.C. *test_file* is a file containing the test vectors, TEST lines long, and the last two parameters are two output files. The first output file, *output_file1* is a log file, which

specifies which class each of the test vectors belongs to. The second output file, *output_file* is very similar to the first, and specifies which class number the test vector corresponding to the line number belongs to. This second file is used as an input to MATLAB so a error counting program can be written to compute errors on the test set.

4.2 Backpropagation Network, Single Hidden Layer, NBP2.C

This program takes a labelled input dataset and a given random weight file and trains a single hidden layer neural network of a specified dimension. The input files consist of a randomly arranged file and corresponding label file, specified in the exact same manner as the outputs of the RANDCOMBINE.C program.

The network is fully connected and specified in the *constants.h* file, included in the code. Specifically, the network is defined as having NIU input units, NHU hidden units, and NOU outputs units, all quantities excluding bias units. In addition, this file allows the user to specify the number of training sets, NUM_SETS, each set consisting of PPS training vectors (batch size). In other words, the training files have to be at least NUM_SETS*PPS lines long. In addition, this file gives the default names for the starting weight file, input pattern file, and input pattern label file. However, all these parameters can be specified at the runline also.

The program provides the user with a simple interactive shell environment in which weight files can be read and written, learning rates and momentum values adjusted, and iterations specified. The interactive shell is implemented using a simple parser table, and is very straightforward. This shell requires an include file, *cparser2.h* to define the various commands. Specifically, the commands are:

```
read_weight_file < weight_file >
write_weight_file < weight_file >
go < iterations >
quit
lrc1= < learning_rate_input_hidden >
lrc2= < learning_rate_hidden_output >
momentum= < momentum_value >
epoch= < display_epoch_number >
```

The purpose of each of these commands is very straightforward. The first two read and write weight files. The "go" command makes the program run a specified number of iterations, with each iteration passing the whole data set through (ie. NUM_SETS batches). The weights are adjusted on every batch,

PPS training vectors, and the error echoed to the screen after all batches are passed (iteration). The "quit" command allows the user to exit the shell back into the *Unix* environment, with the option of saving the weights. "lrc1=", "lrc2=", and "momentum=" allow the user to adjust network parameters during training. Note that since the command parser in the interactive shell is very simple, all commands have to be typed in exactly as specified. For example, the read and write weight file commands, the file name should be separated by only one space from the end of the command. Similarly, to adjust learning rates or momentum values, no spaces are allowed either side of the "=" sign.

The program basically reads in the training data and initial weight files and sets the learning rates and momentum values. If the user does not specify values for these inputs in the runline, default values are used. The default learning rates are both 0.1 and the momentum 0.5. Next, for each batch, a forward pass is computed, the error summed, and a backward weight update performed. Intermediate quantities computed in the backpropagation from the hidden-output layer are stored and used to compute values for the input-hidden layer. After all training batches are presented, the error is displayed. The code is fully commented and the runline for the program is:

```
nbp2 -l lrc1 lrc2 -m momentum -i input_file -o label_file -w weight_file
```

where *lrc1* and *lrc2* are the learning rate constants for the input-hidden and hidden-output layers respectively and *momentum* is the momentum value. *input_file* is the randomly rearranged training vector file, and *label_file* is the corresponding label file, outputs of RANDCOMBINE.C, and *weight_file* is the weight file. The format of the weight file is fully described in the commented code.

4.3 Backpropagation Network, Double Hidden Layer, NBP3.C

This program is exactly the same as NBP2.C except that code has been added to handle an additional hidden layer. The training files used are exactly the same, and the interactive shell has one extra command over that for the single hidden layer program, "lrc3=", to specify the learning rate

constant for the hidden2-output layer. The code's functionality is also very similar to that of NBP2.C, where a forward pass is computed, error summed, and then a backward pass done to update weights, with intermediate results being stored. The runline for the program is identical to that for the single hidden layer, with a slight modification made for the extra learning rate:

```
nbp3 -l lrc1 lrc2 lrc3 -m momentum -i input_file -o label_file -w weight_file
```

where *lrc1*, *lrc2*, and *lrc3* are the learning rates for the input-hidden1, hidden1-hidden2, and hidden2-output layers respectively.

4.4 Optimal Brain Damage, Single Hidden Layer, NBP2OBD.C

This program takes a labelled input dataset and a given weight file and computes the saliencies for the weights in a manner specified in the optimal brain damage (OBD) method. The format for the input and weight files is exactly the same as in that for the straight single hidden layer backpropagation network (NBP2.C).

The interactive shell for this program contains all the commands available for NBP2.C and some additional ones for the OBD functions. Specifically, the additions are:

```
write_dd_file < double_derivative_file >
write_saliency_file < saliency_file >
obd
thresh= < threshold >
decay= < weight_decay >
```

where "write_dd_file" writes the double derivatives for the weights (format identical to weight file), and "write_saliency" writes the saliencies of the weights, computed by using the double derivative information. "thresh=" specifies the threshold saliency magnitude below which all weights are decayed by a factor of *weight_decay*. This decay term can be varied by using the "decay=" command. The "obd" command actually decays the weights by comparing their saliencies against *threshold*.

The functionality for the code again has the same overall setup as that for the straight backpropagation algorithm. First a forward pass is computed,

and error terms summed. On the backward pass, however, both first and second derivatives of the weights are computed, and intermediate results stored to speed up computations in embedded layers. Again, the double derivatives are computed over each batch size, PPS vectors, and then averaged over the whole training set (ie. NUMSETS batches) to get a smoothed estimate of the double derivative. The code for this program is fully documented and all routines are fully explained. Note that since originally, this code is derived by modifying the code for the straight single hidden layer backpropagation network, the runline is identical to that for NBP2.C. As a result, the user has to set the threshold and decay constants in the interactive shell using the various commands. The default values for these constants are thresh=0.0001 and decay=0.8. In the research, the weights are zeroed (decay=0.0) since the decaying procedure is very time consuming. Double derivative information from a given iteration is overwritten on each subsequent iteration through the dataset. This is because there is no momentum term involved for the second derivatives, as is the case with the first derivatives for training purposes. Since the iteration time for this OBD is now much more than for the straight forward single hidden layer network, it is assumed that *weight_file* represents a fully trained network configuration at a good enough minima. Hence only one iteration needs to be done to compute the saliencies (ie. "go 1").

4.5 Optimal Brain Damage, Double Hidden Layer, NBP3OBD.C

This program is exactly the same as NBP2OBD.C except that code has been added to handle an additional hidden layer. The training files used are exactly the same, and the interactive shell has one extra command over that for the single hidden layer program, "lrc3=", to specify the learning rate constant for the hidden2-output layer. The code's functionality is also very similar to that of NBP2OBD.C, where a forward pass is computed, error summed, and then a backward pass done to update weights and compute double derivatives, with intermediate results being stored. The runline for the program is identical to that for the straight double hidden layer backpropagation network. The threshold and decay constants have to be set by the user by using the various shell commands, and the defaults values are

the same as those for the single hidden layer OBD. Running OBD for this double hidden layer case is identical to running it for the single hidden layer case, specified in the previous section.

4.6 Testing Programs: FWDBP2.C, FWDBP3.C

Programs to compute simple forward passes on vector files also exist. These programs are very simple and simply read in the specified test and weight files and compute a forward pass, writing the output neurons activations to a specified filename. The code for these programs is fully commented and the runline for both single and double hidden layer networks is:

```
fwdbp2 -w weight_file -i input_file -o output_file
```

where *weight_file* is the weight file corresponding to the network size specified in *constants.h*, *input_file* is the input testing file, and *output_file* the output activation file.

Program Listings


```

.....
/* Program: PSD.C */
/* Written by: Nasir Ghami */
/* Purpose: To compute the PSD-averaged periodogram for a given */
/*           input signal. The averaging done on the periodogram is */
/*           specified by a filter set file, whose format is: */
/*           */
/*           number of filters */
/*           filter1 start bin, filter1 end bin */
/*           filter2 start bin, filter2 end bin */
/*           . */
/*           filterN start bin, filterN end bin */
/*           */
/*           The filter file has to be manually computed and entered */
/*           before this program can be run. The sample rate, and */
/*           FFT size have to be taken into account to translate the */
/*           required frequency range of the filters into start and */
/*           end bin values. */
/* Runline: psd <input_file> <output_file> <filter_set_file> */
.....
#include <stdio.h>
#include <math.h>

#define SAMPLE_SIZE 8192 /* FFT Size */
#define N2PI 6.2831853 /* 2 PI */
#define MAX_FILTERS 100 /* Maximum number of filters, can be set
                          according to users specifications, must
                          be at least greater than number of filters
                          specified in the filter_set_file file */
#define MID_POINT 1638 /* Bin Number around which to center the filter,
                          ie. this is the bin corresponding to the
                          IF, must be manually computed by taking the
                          FFT size into account:
                          MID_POINT = IF * SAMPLE_SIZE * Ts
                          where IF is Intermediate frequency, in Hz,
                          and Ts is the sample period. This has to be
                          specified before compiling and running */

void get_frame();
void fouri();
void reallf();
void do_hamming();
void adjust();
void read_filters();

main(argc,argv)
int argc;
char *argv[];

{
    int i,n,isign,num_filters;
    float data[SAMPLE_SIZE+1]; /* Array containing data, first location, data[0]
                                is not used by FFT routine */
    int filter_array[MAX_FILTERS][2]; /* Filter Array, specifying the start and
                                        stop bins for each filter */
    FILE *in_file,
          *out_file,
          *filters_file;

    /* First
       data[0]=0.0;

    /* Open all the files */
    in_file = fopen(argv[1],"r");
    out_file = fopen(argv[2],"w");
    filters_file = fopen(argv[3],"r");

    /* Read in filter set information to filter_array */
    read_filters(filters_file,&num_filters,filter_array);

    /* Close filter_set_file */
    fclose(filters_file);

    n=SAMPLE_SIZE/2;
    isign=1;

    /* Load data into array */
    get_frame(in_file,data);

```

```

/* Hamming window data */
do_hamming(data);

/* Compute FFT of data using routine in NUMERICAL RECIPES IN C */
/* Output is complex, X(k), k=0,...,N-1,
   and stored in the in array data also, arrangement is:
   data[1] = Re {X[0]}, Im{X[0]}=0 for real input data
   data[2] = X[N/2], Im{X[N/2]} for real input data
   data[3] = Re {X[1]}
   data[4] = Im {X[1]}
   data[5] = Re {X[2]}
   data[6] = Im {X[2]}

   data[SAMPLE_SIZE-1] = Re {X[(N/2)-1]}
   data[SAMPLE_SIZE] = Im {X[(N/2)-1]} */

realft(data,n,sign);

/* Average Periodogram using filter set and write out results */
adjust(out_file,data,num_filters,filter_array);

}
/*****/
void get_frame(in_file,data)

FILE *in_file; /* File containing data */
float data[]; /* Array into which data is read */
{
    int i;

    /* Loop over SAMPLE_SIZE, and read in data */
    for (i=1;i<=SAMPLE_SIZE;i++)
    {
        fscanf(in_file,"%f",&data[i]);
    };
}

/*****/
void do_hamming(data)

float data[]; /* Array of length SAMPLE_SIZE which is to be weighted with
               a Hamming window of length SAMPLE_SIZE */

{
    int i;

    /* Loop over all points and weight with appropriate Hamming window value */
    for (i=1;i<=SAMPLE_SIZE;i++)
    {
        data[i]=data[i]*(0.54-0.46*cos((N2PI*(i-1))/(SAMPLE_SIZE-1)));
    }
}
/*****/
void adjust(out_file,data,num_filters,filter_array)

FILE *out_file; /* output file */
float data[]; /* File containing complex FFT of original time data */
int num_filters; /* Number of filters */
int filter_array[MAX_FILTERS][2]; /* Array specifying start and stop bin
                                   of each filter */

{
    int i,j,k;
    float tmp,sum;
    norm, /* Magnitude Normalizing Factor */
    temp;
    float *temp_array,*temp_ptr; /* Temporary variables */
    int tmp1,tmp2;

    /* First go through data[] array and compute magnitude of the complex values,
       NOTE the structure of the output described earlier. Now the magnitudes of
       the complex values ranging from data[1] to data[SAMPLE_SIZE] will be stored
       in data[1] to data[SAMPLE_SIZE/2] */

    if(data[1]<0)
    {
        data[1]=(-1.0*data[1]);
    }
}

```

```

/* Save data[2], ie corresponding to real (N/2)-th FFT bin value. */
tmp=data[2];

/* For other bins, whose components have imaginary components, compute
magnitude and store in appropriate bin location */
for (i=3;i<=SAMPLE_SIZE-1;i+=2)
{
    k=(i+1)/2;
    data[k]=((data[i]*data[i])+(data[i+1]*data[i+1]));
}

/* Append last bin value, corresponding to sample_frequency/2, the maximum
value permitted by the Nyquist Theorem */
if (tmp<0.0)
{
    data[(SAMPLE_SIZE/2)+1]=(-1.0*tmp);
}
else
{
    data[(SAMPLE_SIZE/2)+1]=tmp;
}

/***** Output Processing *****/
/* Compute the output for each filter */

/* Zero the normalizing factor */
norm=0.0;

/* Assign temporary storage and pointer for each filter */
temp_array=(float *) malloc(num_filters*sizeof(float));
temp_ptr=temp_array;

/* Loop over filters, sum contents over bin range */
for (i=0;i<num_filters;i++)
{
    /* tmp1=start bin of filter[i], tmp2=end bin of filter[i] */
    tmp1=filter_array[i][0];
    tmp2=filter_array[i][1];

    /* Zero sum */
    sum=0.0;

    /* Loop and accumulate PSD-averaged sum */
    for(k=tmp1;k<tmp2;k++)
    {
        sum=sum+(data[k]*data[k]);
    };
    /* Increment norm for cumulative sum over all filters */
    norm+=sum;
    /* Increment pointer to next, and take square root of sum */
    *temp_ptr+=sqrt(sum);
};

norm=sqrt(norm);

/* Normalize and print to output file */
temp_ptr=temp_array;
for (i=0;i<num_filters;i++)
{
    sum=(*temp_ptr)/norm;
    fprintf(out_file,"%f",sum);
};

fprintf(out_file,"\n");

/* Free temporary variables */
free(temp_array);
free(temp_ptr);
}

/*****/
void read_filters(filters_file,num_filters,filter_array)

FILE *filters_file; /* Filter specification file */
int *num_filters; /* Number of filters */
int filter_array[MAX_FILTERS][2]; /* Array specifying start and stop bin
of each filter */
{

```

```

int i,j;

fscanf(filters_file,"%d",&num_filters); /* Read in the number of filters */

/* Loop over all filters, and read in specified ranges for each one */
for (i=0;i<(*num_filters);i++)
{
    fscanf(filters_file,"%d",&filter_array[i][0]);
    fscanf(filters_file,"%d",&filter_array[i][1]);
};
}

NOTE: All following routines are extracted from NUMERICAL RECIPES
      IN C: THE ART OF SCIENTIFIC COMPUTING, Cambridge University
      Press, New York 1988, pp. 416-418.
      The interested reader is referred to this text for detailed descriptions
      on their functionality
...../
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void fourl(data,nn,isign)
float data[];
int nn,isign;
{
    int n,mmax,m,j,istep,i;
    double wtemp,wr,wpr,wpi,wi,theta;
    float tempr,tempi;

    n=nn << 1;
    j=1;
    for (i=1;i<nn;i+=2) {
        if (j > i) {
            SWAP(data[j],data[i]);
            SWAP(data[j+1],data[i+1]);
        }
        m=n >> 1;
        while (m >= 2 && j > m) {
            j <= m;
            m >>= 1;
        }
        j += m;
    }
    mmax=2;
    while (n > mmax) {
        istep=2*mmax;
        theta=6.28318530717959/(isign*mmax);
        wtemp=sin(0.5*theta);
        wpr = -2.0*wtemp*wtemp;
        wpi=sin(theta);
        wr=1.0;
        wi=0.0;
        for (m=1;m<mmax;m+=2) {
            for (i=m;i<nn;i+=istep) {
                j=i+mmax;
                tempr=wr*data[j]-wi*data[j+1];
                tempi=wr*data[j+1]+wi*data[j];
                data[j]=data[i]-tempr;
                data[j+1]=data[i+1]-tempi;
                data[i] += tempr;
                data[i+1] += tempi;
            }
            wr=(wtemp*wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
        }
        mmax=istep;
    }
}

#undef SWAP
...../

void reallt(data,n,isign)
float data[];
int n,isign;
{
    int i,i1,i2,i3,i4,n2p3;
    float c1=0.5,c2,h1r,h1i,h2r,h2i;
    double wr,wi,wpr,wpi,wtemp,theta;

```

```

void four1()
{
    theta=3.141592653589793/(double) n;
    if (isign == 1) {
        c2 = -0.5;
        four1(data,n,1);
    } else {
        c2=0.5;
        theta = -theta;
    }
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0+wpr;
    wi=wpi;
    n2p3=2*n+3;
    for (i=2;i<=n/2;i++) {
        i4=1+(i3=n2p3-(i2=1+(i1=i+1)));
        h1r=c1*(data[i1]+data[i3]);
        h1i=c1*(data[i2]-data[i4]);
        h2r = -c2*(data[i2]+data[i4]);
        h2i=c2*(data[i1]-data[i3]);
        data[i1]=h1r+wr*h2r-wi*h2i;
        data[i2]=h1i+wr*h2i+wi*h2r;
        data[i3]=h1r-wr*h2r+wi*h2i;
        data[i4] = -h1i+wr*h2i+wi*h2r;
        wr=(wtemp-wr)*wpr-wi*wpi+wr;
        wi=wi*wpr+wtemp*wpi+wi;
    }
    if (isign == 1) {
        data[1] = (h1r-data[1])+data[2];
        data[2] = h1r-data[2];
    } else {
        data[1]=c1*((h1r-data[1])+data[2]);
        data[2]=c1*(h1r-data[2]);
        four1(data,n,-1);
    }
}

```

```

/*****
/* Program: PER.C                               */
/* Written by: Nasir Ghani                       */
/* Purpose: To compute the modified Welch periodogram for a given */
/*           input signal. The SAMPLE_SIZE is equal to the FFT size */
/*           chosen and SAMPLES specifies the total number of points. */
/*           Therefore the input file has to have at least SAMPLES */
/*           number of points and thus the number of averages taken */
/*           to generate the spectrum estimate is: */
/*           K=SAMPLES/SAMPLE_SIZE */
/*           In addition, the user has to specify the bins */
/*           corresponding to the lower and upper bandwidths in about */
/*           the IF of the signal (HIGHER, LOWER constants). The IF */
/*           itself has to be specified in a manner exactly the same */
/*           as that in the PSD.C program. The program breaks up the */
/*           input sample stream into frames, computes FFT's for each */
/*           and keeps a running average of the frames. To speed up */
/*           runtime for overlapped frames, two data frames are */
/*           specified, and overlapping data copied into both frames */
/*           to reduce run time. */
/* Runline: per <input_file> <output_file> */
*****/

#include <stdio.h>
#include <math.h>

#define SAMPLES      8192 /* Total number of sample points */
#define SAMPLE_SIZE  512 /* FFT size */
#define SEPARATION    512 /* Separation in samples between frames */
#define OVERLAP      0 /* Overlap between frames, */
                        /* SAMPLE_SIZE=OVERLAP+SEPARATION */
#define NZPI          6.2831853 /* 2 Pi */
#define HIGHER        137 /* Bin corresponding to lower edge of BW */
#define LOWER         68 /* Bin corresponding to higher edge of BW */

void get_frame();
void normalize_frame();
void fourl();
void reallf();
void do_hanning();
void adjust();
void write_mean();

main(argc,argv)
int argc;
char *argv[];

{
    int i,n,isign, /* Temporary variables */
        count, /* Keeps track of number of frames processed */
        num_frames; /* Number of frames over all SAMPLES */

    float data1[SAMPLE_SIZE+1], /* First data frame */
          data2[SAMPLE_SIZE+1], /* Second data frames, contains OVERLAP samples
                                from data1 */
          mean1[HIGHER-LOWER+1]; /* Running sum over all processed frames of
                                the desired bandwidth range */
    FILE *in_file,*out_file; /* input and output file pointers */

    /* Zero unused portion of data array */
    data1[0]=0.0;
    data2[0]=0.0;

    /* Open files */
    in_file = fopen (argv[1],"r");
    out_file = fopen (argv[2],"w");

    /* Read in first OVERLAP samples in data1[], starts of the alternating data1[]
    data2[] reading done by GET_FRAME routine */
    for (i=1;i<OVERLAP;i++)
    {
        fscanf(in_file,"%f",&data1[i]);
    }

    /* Zero running sum variable */
    for (i=LOWER;i<=HIGHER;i++)

```

```

{
    mean1[i-LOWER]=0.0;
};

/* Specify input to FFT routine */
n=SAMPLE_SIZE/2;
isign=1;

/* Zero count */
count=0;

/* Compute number of frames to process, each from SAMPLE_SIZE long */
num_frames = (int) (SAMPLES/SEPARATION);

/* Loop over all frames */
while (count<num_frames)
{
    /* Read frame into data1[], copy overlapping part towards end of this
       frame into beginning of data2[] */
    get_frame (in_file,data1,data2);

    /* Hamming weight frame */
    do_hamming(data1);

    /* Compute FFT of data using routine in NUMERICAL RECIPIES IN C */
    /* Output is complex, X(k), k=0,...,N-1,
       and stored in the in array data also, arrangement is:
       data[1] = Re {X[0]}, Im {X[0]}=0 for real input data
       data[2] = X[N/2],   Im {X[N/2]} for real input data
       data[3] = Re {X[1]}
       data[4] = Im {X[1]}
       data[5] = Re {X[2]}
       data[6] = Im {X[2]}

       data[SAMPLE_SIZE-1] = Re {X[ (N/2)-1 ]}
       data[SAMPLE_SIZE]  = Im {X[ (N/2)-1 ]} */

    realft(data1,n,isign);

    /* Extract desired bandwidth from FFT results, and update running mean */
    adjust(data1,mean1);

    /* Increment count, check if not exceeded, if not, read in another frame
       and this time data and processing is done with data2[], ie. roles between
       data1[] and data2[] reversed to prepare for other iteration of while
       loop */
    count++;

    if (count<num_frames)
    {
        /* Read frame into data2[], copy overlapping part towards end of this
           frame into beginning of data1[] */
        get_frame (in_file,data2,data1);

        /* Hamming weight frame */
        do_hamming(data2);

        /* Compute FFT of frame, result in data2[], in manner specified in
           NUMERICAL RECIPIES IN C routine */
        realft(data2,n,isign);

        /* Extract desired bandwidth from FFT results, and update running mean */
        adjust(data2,mean1);

        /* Increment frames-processed count */
        count++;
    }
}

/* Write out results to output file */
write_mean(mean1,num_frames,out_file);
}

/*****
void get_frame(in_file,data1,data2)

```



```

FILE *in_file; /* Input file */
float data1[], data2[]; /* SAMPLE_SIZE samples read into data1[], and
OVERLAP samples corresponding to overlap between
current frame (data1[]) and next frame (data2[])
also copied to data2[] to save time */

{
int i;

/* Read SEPARATION values into data1[] to completely fill what remains of it.
ie. OVERLAP samples read on last cycle */
for (i=OVERLAP+1; i<=SAMPLE_SIZE; i++)
{
fscanf(in_file, "%f", &data1[i]);
}

/* Copy overlap between data1[] and data2[] into data2[] */
for (i=SEPARATION+1; i<=SAMPLE_SIZE; i++)
{
data2[i-SEPARATION]=data1[i];
}
}

/*****
void do_hamming(data)

float data[]; /* Input frame to be Hamming windowed */

{
int i;

/* Compute Hamming window the frame, depending on SAMPLE_SIZE */
for (i=1; i<=SAMPLE_SIZE; i++)
{
data[i]=data[i]*(0.54-0.46*cos((N2PI*(i-1))/(SAMPLE_SIZE-1)));
}
}
*****/

void adjust(data, mean1) /* Update running sum over all frames */
float data[], /* Frame containing data in RECIPIES IN C format,
specified above */
mean1[]; /* Running sum */

{
int i, j;
j=0;

/* Compute magnitudes and update average. Note that the code assumes that
the bandwidth (specified by HIGHER and LOWER) does NOT extend down to
dc (0 Hz, ie LOWER=0) and up to the maximum Nyquist rate
(ie. sampling frequency/2, HIGHER=SAMPLE_SIZE/2) */
for (i=(2*LOWER+1); i<=(2*HIGHER+1); i+=2)
{
mean1[j]+=(data[i]*data[i]+data[i+1]*data[i+1]);
j++;
}
};

/*****
void write_mean(mean1, num_frames, out_file)
float mean1[]; /* Overall mean over all frames for desired bandwidth */
int num_frames; /* Total number of frames processed */
FILE *out_file; /* Output file pointer */

{
int i;
float norm, /* Normalizing constant */
temp; /* Temporary variable */

/* Initialize norm */
norm=0.0;

/* Compute norm over accumulated sum */
for (i=LOWER; i<=HIGHER; i++)
{
norm+=mean1[i-LOWER]*mean1[i-LOWER];
}
}

```

```

norm=sqrt(norm);

/* Write out results, normalizing each value */
for (i=LOWER;i<=HIGHER;i++)
{
    fprintf(out_file,"%f ",(mean1[i-LOWER]/norm));
};

fprintf(out_file,"\n");

};

/*****
NOTE: All following routines are extracted from NUMERICAL RECIPES
      IN C: THE ART OF SCIENTIFIC COMPUTING, Cambridge University
      Press, New York 1988, pp. 416-418.
      The interested reader is referred to this text for detailed descriptions
      on their functionality
*****/
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void fourl(data,nn,isign)
float data[];
int nn,isign;
{
    int n,mmax,m,j,istep,i;
    double wtemp,wr,wpr,wpi,wi,theta;
    float tempr,tempi;

    n=nn << 1;
    j=1;
    for (i=1;i<nn;i+=2) {
        if (j > i) {
            SWAP(data[j],data[i]);
            SWAP(data[j+1],data[i+1]);
        }
        m=n >> 1;
        while (m >= 2 && j > m) {
            j -= m;
            m >>= 1;
        }
        j += m;
    }
    mmax=2;
    while (n > mmax) {
        istep=2*mmax;
        theta=6.28318530717959/(isign*mmax);
        wtemp=sin(0.5*theta);
        wpr = -2.0*wtemp*wtemp;
        wpi=sin(theta);
        wr=1.0;
        wi=0.0;
        for (m=1;m<mmax;m+=2) {
            for (i=m;i<nn;i+=istep) {
                j=i+mmax;
                tempr=wr*data[j]-wi*data[j+1];
                tempi=wr*data[j+1]+wi*data[j];
                data[j]=data[i]-tempr;
                data[j+1]=data[i+1]-tempi;
                data[i] += tempr;
                data[i+1] += tempi;
            }
            wr=(wtemp*wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
        }
        mmax=istep;
    }
}

#undef SWAP

/*****
void realft(data,n,isign)
float data[];
int n,isign;
{
    int i,j1,j2,i3,j4,n2,p3;
    float c1=0.5,c2,h1r,b1i,h2r,h2i;

```

```

double wr,wi,wpr,wpi,wtemp,theta;
void fourlQ;

theta=3.141592653589793/(double)n;
if (isign == 1) {
    c2 = -0.5;
    fourl(data,n,1);
} else {
    c2=0.5;
    theta = -theta;
}
wtemp=sin(0.5*theta);
wpr = -2.0*wtemp*wtemp;
wpi=sin(theta);
wr=1.0+wpr;
wi=wpi;
n2p3=2*n+3;
for (i=2;i<=n/2;i++) {
    i4=i+(i3=n2p3-(i2=1+(i1=i-i-1)));
    h1r=c1*(data[i1]+data[i3]);
    h1i=c1*(data[i2]-data[i4]);
    h2r = -c2*(data[i2]+data[i4]);
    h2i=c2*(data[i1]-data[i3]);
    data[i1]=h1r+wr*h2r-wi*h2i;
    data[i2]=h1i+wr*h2i+wi*h2r;
    data[i3]=h1r-wr*h2r+wi*h2i;
    data[i4] = -h1i+wr*h2i-wi*h2r;
    wr=(wtemp-wr)*wpr-wi*wpi+wr;
    wi=wi*wpr+wtemp*wpi+wi;
}
if (isign == 1) {
    data[1] = (h1r-data[1])+data[2];
    data[2] = h1r-data[2];
} else {
    data[1]=c1*((h1r-data[1])+data[2]);
    data[2]=c1*(h1r-data[2]);
    fourl(data,n,-1);
}
}
/...../

```

```

/*****
/* Program: BSP.C
/* Written by: Nasir Ghani
/* Purpose: To compute the bispectrum of an inputted time signal.
/* The SAMPLE_SIZE is equal to the FFT size
/* chosen and SAMPLES specifies the total number of points.
/* Therefore the input file has to have at least SAMPLES
/* number of points and thus the number of averages taken
/* to generate the spectrum estimate is:
/*
/* K=SAMPLES/SAMPLE_SIZE
/*
/* The program breaks up the
/* input sample stream into frames, computes bispectra for
/* each, keeping a running average over frames. To speed
/* up runtime for overlapped frames, two data frames are
/* specified, and overlapping data copied into both frames
/* to reduce run time.
/*
/* Runline: bsp <input_file> <output_file>
*****/
#include <stdio.h>
#include <math.h>

#define SAMPLES 8192 /* Total number of sample points */
#define SAMPLE_SIZE 64 /* FFT size */
#define SEPARATION 64 /* Separation in samples between frames */
#define OVERLAP 0 /* Overlap between frames,
SAMPLE_SIZE=OVERLAP+SEPARATION */
#define N2PI 6.2831853 /* 2 Pi */
#define OPT_CON_2 0.1013211 /* Constant for optimal bispectral window,
sqrt(3)/(Pi * Pi) */

void get_frame();
void fourl();
void realft();
void adjust();
void opt_win();
void write_bspn();

main(argc,argv)
int argc;
char *argv[];

{
    int i,j,n,sign, /* Temporary variables */
        count, /* Keeps track of number of frames processed */
        num_frames; /* Total Number of frames to be processed */
    float data1[SAMPLE_SIZE+1], /* First frame */
        data2[SAMPLE_SIZE+1]; /* Second frame, contains OVERLAP samples from
data1 */
    float bspr[SAMPLE_SIZE/2+1][SAMPLE_SIZE/2+1][2]; /* Running sum of bispectrum
over all frames, complex, bspr[][][0]=Real
bspr[][][1]=Complex */
    float bspn[SAMPLE_SIZE/2+1][SAMPLE_SIZE/2+1]; /* Magnitude of bispectrum,
scalar */
    FILE *in_file,*out_file; /* Input and output file pointers */

    /* Zero unused locations of data arrays */
    data1[0]=0.0;
    data2[0]=0.0;

    /* Zero sums for bispectrum estimate */
    for (i=1;i<=(SAMPLE_SIZE/2);i++)
    {
        for (j=1;j<=(SAMPLE_SIZE/2);j++)
        {
            bspr[i][j][0]=0.0;
            bspr[i][j][1]=0.0;
            bspn[i][j]=0.0;
        }
    }

    /* Open files */
    in_file = fopen (argv[1],"r");
    out_file = fopen (argv[2],"w");

```

```

/* Read in first OVERLAP samples in data1[], starts of the alternating data1[]
data2[] reading done by GET_FRAME routine */
for (j=1;j<OVERLAP;j++)
{
    fscanf(in_file,"%f",&data1[j]);
}

/* Specify input to FFT routine */
n=SAMPLE_SIZE/2;
isign=1;

/* Zero frame-processed count */
count=0;

/* Compute number of frames processed */
num_frames = (int) (SAMPLES/SEPARATION);

/* Loop over all frames */
while (count<num_frames)
{
    /* Read frame into data1[], copy overlapping part towards end of this
    frame into beginning of data2[] */
    get_frame (in_file,data1,data2);

    /* Compute FFT of data using routine in NUMERICAL RECIPES IN C */
    /* Output is complex, X(k), k=0,...,N-1,
    and stored in the in array data also, arrangement is:
    data[1] = Re {X[0]}, Im {X[0]}=0 for real input data
    data[2] = X[N/2], Im {X[N/2]} for real input data
    data[3] = Re {X[1]}
    data[4] = Im {X[1]}
    data[5] = Re {X[2]}
    data[6] = Im {X[2]}

    data[SAMPLE_SIZE-1] = Re {X[(N/2)-1]}
    data[SAMPLE_SIZE] = Im {X[(N/2)-1]} */
    realft(data1,n,isign);

    /* Using FFT, compute bispectrum for the frame, and update running sum,
    contained in bspr[1][1]. Method used is that specified by Nikias and
    Raghuveer in paper BISPECTRUM ESTIMATION: A DIGITAL SIGNAL PROCESSING
    FRAMEWORK. */
    adjust(data1,bspr);

    /* Increment count, check if not exceeded, if not, read in another frame
    and this time data and processing is done with data2[], ie. roles between
    data1[] and data2[] reversed to prepare for other iteration of while
    loop */
    count++;

    if (count<num_frames)
    {
        /* Read frame into data2[], copy overlapping part towards end of this
        frame into beginning of data1[] */
        get_frame (in_file,data2,data1);

        /* Compute FFT of frame, result in data2[], in manner specified in
        NUMERICAL RECIPES IN C routine */
        realft(data2,n,isign);

        /* Using FFT, compute bispectrum for the frame, and update running sum,
        contained in bspr[1][1]. Method used is that specified by Nikias and
        Raghuveer in paper BISPECTRUM ESTIMATION: A DIGITAL SIGNAL PROCESSING
        FRAMEWORK. */
        adjust(data2,bspr);

        /* Increment frame-processed count */
        count++;
    }
}

/* Compute magnitude of bispectrum from complex sum over all frames */
/* Remove this loop and adjust write_bpsm() routine IF complex output required */
for (i=1;i<=(SAMPLE_SIZE/2);i++)
{
    for (j=1;j<=(SAMPLE_SIZE/2);j++)
    {

```

```

    bspm[i][j]=sqrt(bspm[i][j][0]*bspm[i][j][0] + bspm[i][j][1]*bspm[i][j][1]);
  };
};

/* Optimally window magnitude of bispectrum */
opt_win(bspm);

/* Write bispectral magnitude to output file */
write_bspm(bspm,num_frames,out_file);

}

/*****
void get_frame(in_file,data1,data2)

FILE *in_file; /* Input file */
float data1[],data2[]; /* SAMPLE_SIZE samples read into data1[], and
OVERLAP samples corresponding to overlap between
current frame (data1[]) and next frame (data2[])
also copied to data2[] to save time */

{
  int i;

  /* Read SEPARATION values into data1[] to completely fill what remains of it..
  ie. OVERLAP samples read on last cycle */
  for (i=OVERLAP+1;i<=SAMPLE_SIZE;i++)
  {
    fscanf(in_file,"%f",&data1[i]);
  }

  /* Copy overlap between data1[] and data2[] into data2[] */
  for (i=SEPARATION+1;i<=SAMPLE_SIZE;i++)
  {
    data2[i-SEPARATION]=data1[i];
  }
}

*****/
void adjust(data,bspr) /* Update sum (bspr[][][]) with bispectrum of
latest frame, whose FFT is in data. */

float data[], /* FFT of data frame */
bspr[SAMPLE_SIZE/2+1][SAMPLE_SIZE/2+1][2]; /* Complex running sum array,
to be updated */

{
  int i,j,i1,j2,k1,k2; /* Variables for indexing into arrays */
  float a,b,c,d,e,f,temp1,temp2,tmp1;

  /* Since (N/2)-th value of FFT is in data[2], store it in tmp1, and zero
  that location so that the i-th FFT value can be retrieved by simply
  using the formula
    Re{X(i)} = data[2*i-1]
    Im{X(i)} = data[2*i]
  This avoids having if conditions in the loops to check if i=1, and hence
  speeds up the program. However, to handle bispectrum values requiring
  the (N/2)-th FFT location, a loop is added at the end. This inconvenience
  is due to the storage scheme for FFT results used in the NUMERICAL RECIPIES
  IN C routine */

  tmp1=data[2];
  data[2]=0.0;

  /* Cover region between w1>=0, w2<=(N/4), w1+w2=(N/2) */
  for (i=1;i<=(SAMPLE_SIZE/4);i++)
  {
    for (j=1;j<=i;j++)
    {
      /* Compute X(w1) */
      i1=2*i-1;
      a=data[i1];
      b=data[i1+1];

      /* Compute X(w2) */
      j1=2*j-1;
      c=data[j1];
      d=data[j1+1];

      /* Compute X(w1+w2) */
      k1=2*(i+j)-1;

```

```

    c=data[k1];
    f=-1.0*data[k1+1];

    /* Compute Re and Im parts of X(w1)X(w2)X(w1+w2) */
    temp1=(a*c-b*d)*e+(b*c+d*a)*f;
    temp2=(b*c+d*a)*e-(a*c-b*d)*f;
    bspr[i][j][0]+=temp1;
    bspr[i][j][1]+=temp2;
  };
};

/* Cover region between w1>=0, w2>=(N/4), w1-w2=(N/2)-1 */
for (i=(SAMPLE_SIZE/4+1);i<=(SAMPLE_SIZE/2);i++)
{
  for (j=1;j<=((SAMPLE_SIZE/2)-i);j++)
  {
    /* Compute X(w1) */
    i1=2*i-1;
    a=data[i1];
    b=data[i1+1];

    /* Compute X(w2) */
    j1=2*j-1;
    c=data[j1];
    d=data[j1+1];

    /* Compute X(w1+w2) */
    k1=2*(i+j)-1;
    e=data[k1];
    f=-1.0*data[k1+1];

    /* Compute Re and Im parts of X(w1)X(w2)X(w1+w2) */
    temp1=(a*c-b*d)*e+(b*c+d*a)*f;
    temp2=(b*c+d*a)*e-(a*c-b*d)*f;
    bspr[i][j][0]+=temp1;
    bspr[i][j][1]+=temp2;
  };
};

/* Compute strip corresponding to line w1-w2=N/2, and w2>=(N/4) and w1>0 */
for (i=(SAMPLE_SIZE/4+1);i<=(SAMPLE_SIZE/2);i++)
{
  /* Since only strip (line) being computed, no loop over j */
  j=(SAMPLE_SIZE/2)-i+1;

  /* Compute X(w1) */
  i1=2*i-1;
  a=data[i1];
  b=data[i1+1];

  /* Compute X(w2) */
  j1=2*j-1;
  c=data[j1];
  d=data[j1+1];

  /* Compute X(w1+w2) */
  k1=2*(i+j)-1;
  e=temp1;
  f=0.0; /* Since Im(X(N/2)) = 0 for real data */

  /* Compute Re and Im parts of X(w1)X(w2)X(w1+w2=(N/2)) */
  temp1=(a*c-b*d)*e+(b*c+d*a)*f;
  temp2=(b*c+d*a)*e-(a*c-b*d)*f;
  bspr[i][j][0]+=temp1;
  bspr[i][j][1]+=temp2;
};
}
/*****/

void opt_win(bspm) /* Optimally window bispectral magnitude estimate */
{
  float bspm[SAMPLE_SIZE/2+1][SAMPLE_SIZE/2+1]; /* Magnitude of bispectrum */
  int i,j;
  float temp;

  /* Go through first bispectral slice, bounded by lines, w1=0, w1+w2=(N/2),
  and w1-w2=(N/2), and weight each point. Window is obtained from Gabr and

```


Rao, LECTURE NOTES IN STATISTICS: AN INTRODUCTION TO BISPECTRAL ANALYSIS
AND BILINEAR TIME SERIES MODELS, Springer Verlag, New York, 1984. */

```
temp=2*3.1415927/(SAMPLE_SIZE);

/* First cover region between w1>=0, and w2<=(N/4) and w1+w2<=(N/2) */
for (i=1;i<=(SAMPLE_SIZE/4);i++)
{
    for (j=1;j<=i;j++)
    {
        bspn[i][j] *= (1 - OPT_CON_2*temp*temp*( (i*i) + (j*j) + (i*j) ));
    };
};

/* Cover region between w1>=0, and w2>(N/4) and w1-w2<=(N/4) */
for (i=(SAMPLE_SIZE/4)+1;i<=(SAMPLE_SIZE/2);i++)
{
    for (j=1;j<=(SAMPLE_SIZE/2)-i+1;j++)
    {
        bspn[i][j] *= (1 - OPT_CON_2*temp*temp*( (i*i) + (j*j) + (i*j) ));
    };
};

};

/*****

void write_bspn(bspn,num_frames,out_file) /* Write bispectral magnitude */
float bspn[SAMPLE_SIZE/2+1][SAMPLE_SIZE/2+1]; /* Bispectral Magnitude */
int num_frames;
FILE *out_file;

{
    int i,j,k,start,stop;
    float norm,temp;

    /* Normalize with largest value */

    /* Find largest magnitude */
    norm=-100;
    for (i=1;i<=(SAMPLE_SIZE/2);i++)
    {
        for (j=1;j<=(SAMPLE_SIZE/2);j++)
        {
            temp=bspn[i][j];
            if (temp>norm)
            {
                norm=temp;
            };
        };
    };

    /* norm takes care of sum over all frames. Write out results for each w2
    value, varying w1 */

    start=1;
    stop=(SAMPLE_SIZE/2);
    for (k=1;k<=(SAMPLE_SIZE/4);k++)
    {
        for (i=start;i<=stop;i++)
        {
            fprintf(out_file,"%f ",(bspn[i][k]/norm) );
        };
        start=start+1;
        stop=stop-1;
    };

    fprintf(out_file,"\n");

};

*****/
```

NOTE: All following routines are extracted from NUMERICAL RECIPES
IN C: THE ART OF SCIENTIFIC COMPUTING, Cambridge University
Press, New York 1988, pp. 416-418.
The interested reader is referred to this text for detailed descriptions
on their functionality

```

/*****
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void fourl(data,n,sign)
float data[];
int n,sign;
{
    int n,mmax,m,j,istep,i;
    double wtemp,wr,wpr,wpi,wi,theta;
    float tempr,tempi;

    n=n<<1;
    j=1;
    for (i=1;i<n;i+=2) {
        if (j > i) {
            SWAP(data[j],data[i]);
            SWAP(data[j+1],data[i+1]);
        }
        m=n>>1;
        while (m>=2 && j>m) {
            j<=m;
            m>>=1;
        }
        j+=m;
    }
    mmax=2;
    while (n>mmax) {
        istep=2*mmax;
        theta=6.28318530717959/(sign*mmax);
        wtemp=sin(0.5*theta);
        wpr=-2.0*wtemp*wtemp;
        wpi=sin(theta);
        wr=1.0;
        wi=0.0;
        for (m=1;m<mmax;m+=2) {
            for (i=m;i<n;i+=istep) {
                j=i+mmax;
                tempr=wr*data[j]-wi*data[j+1];
                tempi=wr*data[j+1]+wi*data[j];
                data[j]=data[i]-tempr;
                data[j+1]=data[i+1]-tempi;
                data[i]+=tempr;
                data[i+1]+=tempi;
            }
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
        }
        mmax=istep;
    }
}

#undef SWAP
/*****/

void realft(data,n,sign)
float data[];
int n,sign;
{
    int i,j1,i2,i3,i4,n2p3;
    float c1=0.5,c2,h1r,h1i,h2r,h2i;
    double wr,wi,wpr,wpi,wtemp,theta;
    void fourl();

    theta=3.141592653589793/(double)n;
    if (sign==1) {
        c2=-0.5;
        fourl(data,n,1);
    } else {
        c2=0.5;
        theta=-theta;
    }
    wtemp=sin(0.5*theta);
    wpr=-2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0+wpr;
    wi=wpi;
    n2p3=2*n+3;
    for (j=2;j<=n/2;j++) {
        i4=1+(i3=n2p3-(i2=1+(i1=i-j-1)));

```

```

        h1r=c1*(data[i1]+data[i3]);
        h1i=c1*(data[i2]-data[i4]);
        h2r = -c2*(data[i2]+data[i4]);
        h2i=c2*(data[i1]-data[i3]);
        data[i1]=h1r+wr*h2r-wi*h2i;
        data[i2]=h1i+wr*h2i+wi*h2r;
        data[i3]=h1r-wr*h2r+wi*h2i;
        data[i4] = -h1i+wr*h2i+wi*h2r;
        wr=(wtcmp==wr)*wpr-wi*wpi+wr;
        wi=wi*wpr+wtcmp*wpi+wi;
    }
    if (isign == 1) {
        data[1] = (h1r-data[1])+data[2];
        data[2] = h1r-data[2];
    } else {
        data[1]=c1*((h1r-data[1])+data[2]);
        data[2]=c1*(h1r-data[2]);
        fouri(data,n,-1);
    }
}
/...../

```

```

/*****
/* Program: COMBINE.C */
/* Written by: Nasir Ghani */
/* Purpose: To combine individual feature vector files for a given */
/*           signal type into a larger file, according to sequence */
/*           specified in the NUMBERS file */
/* Runline: combine <dir1> <dir2> ... <output_file>
*****/

#include <stdio.h>
#include <string.h>

#define COMBINE 300 /* Number of individual files to be combined */
#define VECTOR 1056 /* Number of features (floating real) per file */
#define SUFFIX ".B128" /* Suffix of individual files */

main (argc,argv)

int argc;
char *argv[];

{
    int i,j,k;
    float x;
    char name1[30], /* Temporary variable */
        numasci[5], /* Temporary variable */
        num_to_string[COMBINE][5]; /* Array containing number to string conversion */
    FILE *NUMFILE, /* file pointer to NUMBERS file */
        *INPUTFILE, /* Temporary file variable */
        *OUTFILE; /* Output file, containing combined files */

    NUMFILE = fopen ("NUMBERS","r");

    /* Read in NUMBERS file to get order and names of files to combine */
    for (i=1;i<=1800;i++)
    {
        fscanf(NUMFILE,"%s",num_to_string[i]);
    }

    fclose(NUMFILE);

    /* Open output file */
    OUTFILE = fopen (argv[argc-1],"w");

    printf("***** WARNING: Combining %d Vector Size *****\n",VECTOR);

    /* Loop over all files to be combined */
    for (j=1;j<=COMBINE;j++)
    {
        /* Loop over all directories, typically, however, only one directory is
        specified since this program is used for single class */
        for (i=1;i<(argc-1);i++)
        {
            /* Generate file name by appending directory, prefix, number, suffix */
            strcpy (name1,argv[i]);
            strcat (name1,"/");
            strcat (name1,num_to_string[j]);
            strcat (name1,SUFFIX);

            /* Open file, read, and copy to output file */
            INPUTFILE = fopen(name1,"r");
            for (k=0;k<VECTOR;k++)
            {
                fscanf(INPUTFILE,"%f",&x);
                fprintf(OUTFILE,"%f",x);
            }
            fclose(INPUTFILE);
            fprintf(OUTFILE,"\n");
        }
    }

    /* Close output file */
    fclose (OUTFILE);
}

```

```

/*****
/* Program:  RANDCOMBINE.C                               */
/* Written by: Nasir Ghani                                */
/* Purpose:  To combine feature files for different signal types into */
/*           a single file in a random order. Outputs an additional */
/*           file, specifying the class of corresponding entries in */
/*           the random file. The need for random ordering and */
/*           labelling of the vectors is required to generate input */
/*           files to the backpropagation software                */
/* Runline:  randcombine <class_file1> <class_file2> ...      */
/*           <rand_out_file1> <rand_out_file2> ...            */
*****/
#include<stdio.h>

#define NUM_CLASS 2 /* Number of classes (files) to combine */
#define NCLASS 1800 /* Number of feature vectors (lines) per class file */
#define VVECTOR 70 /* Size of feature vector */
#define RAND_MAX 2147483647 /* To scale output of random number generator */
#define LTHRESH 0.00005 /* lower threshold, to signify signal does not
                        belong to this class */
#define UTHRESH 0.99995 /* higher threshold, to signify signal does belong
                        to this class */

main(argc,argv)
int argc;
char *argv[];

{
    int i,j,k,
        index, /* temporary variable */
        overall_count; /* Keeps track of how many lines have been combined
                        over all the classes */
    int count_class[NUM_CLASS]; /* Array, keeps track of how many feature vectors
                        (lines) from each class have been combined */
    float out_desired[NUM_CLASS]; /* Temporary vector, to use to label outputs
                        to the rand_out_file1 file, i.e. generate
                        lines of the rand_out_file2 file */
    float temp,temp2,RK;
    char name[30]; /* Temporary string variable */
    FILE *in_file[NUM_CLASS], /* Array, contains the name of the files
                        pertaining to each class */
        *out_file1, /* File pointer to rand_out_file1 */
        *out_file2; /* File pointer to rand_out_file2 */

    /* Open output files */
    out_file1=fopen(argv[argc-2],"w");
    out_file2=fopen(argv[argc-1],"w");

    /* Loop over classes, read in the input file name for each class */
    for (i=0;i<NUM_CLASS;i++)
    {
        in_file[i]=fopen(argv[i+1],"r");
    };

    k=NUM_CLASS*NCLASS; /* Overall number of feature vectors (lines) summed over
                        all classes */
    RK=1.0/NUM_CLASS; /* Floating point value, used in uniform distribution
                        selection */

    /* Zero combined count for all classes */
    for (i=0;i<NUM_CLASS;i++)
    {
        count_class[i]=0;
    };

    /* Zero overall count */
    overall_count=0;

    /* Loop until all feature vectors (lines) in all class files have been
        read and copied to the random output file */
    while(overall_count<k)
    {
        /* Generate random number between 0 and 1 */
        temp=((float)rand())/RAND_MAX;

        /* Compute class, using uniform distribution */
        index=(int)(temp/RK); /* index=class */
    }
}

```

```

/* If all lines for file have not been combined */
if(count_class[index]<NCLASS)
{
    /* Zero labels to all classes first by using lower threshold */
    for (j=0;j<NUM_CLASS;j++)
    {
        out_desired[j]=LTHRESH;
    };
    /* Set desired class to higher threshold */
    out_desired[index]=UTHRESH;

    /* Write out label vector to the rand_out_file2 file, to indicate
       what class the feature vector in the rand_out_file1 file for the
       corresponding line belongs to */
    for (j=0;j<NUM_CLASS;j++)
    {
        fprintf(out_file2,"%d ",out_desired[j]);
    };
    fprintf(out_file2,"\n");

    /* Loop over feature vector size, read class file, and write out to
       the first random file */
    for (j=0;j<VECTOR;j++)
    {
        fscanf(in_file[index],"%d",&temp2);
        fprintf(out_file1,"%d ",temp2);
    };
    fprintf(out_file1,"\n");

    /* Increment combined count for class */
    count_class[index]=count_class[index]+1;
    /* Increment overall combined count */
    overall_count++;
} /* If statement */

};

/* Close class files */
for (i=0;i<NUM_CLASS;i++)
{
    fclose(in_file[i]);
};
}

```

```

/* ***** */
/* Program: KNN.C */
/* Written by: Nasir Ghani */
/* Purpose: K-Nearest neighbour classifier. Takes in test vectors */
/*           and compares them to a database of vectors to find a */
/*           closest match, using the Euclidean distance metric. */
/*           the random file. */
/* Runline: Knn <input_file> <label_file> <test_file> <output_file1> */
/*           <output_file2> */
/*           where input_file and label_file are the training files, */
/*           test_file is the test vector file, and output_file1 and */
/*           output_file2 are the two output files. */
/* ***** */
#include <stdio.h>
#include <string.h>

#define CLASS 10 /* Number of classes, CLASS*TRAIN = FILELENGTH */
#define TRAIN 1800 /* Number of training vectors from each class */
#define TEST 10000 /* Number of overall test vectors */
#define VECTOR 36 /* Dimensionality of training vector */
#define K 1 /* K value, ie. number of nearest neighbours computed */

void get_data();
void get_vector();
void compute_neighbor();
void sort_min();
int get_class();

main(argc,argv)
int argc;
char *argv[];

{
    int i,j,k;
    float data[CLASS][TRAIN][VECTOR], /* Training vectors */
          test_vector[VECTOR]; /* Test vectors */
    char label[CLASS][10]; /* Contains the names of each class, specified by user */
    FILE *input1,*input2,*input3,*output1,*output2; /* Input file pointers */

    /* Declare Class Labels */
    strcpy(label[0],"AM");
    strcpy(label[1],"FM");
    /*strcpy(label[2],"ASK");
    strcpy(label[3],"QPSK");
    strcpy(label[4],"SSBU");
    strcpy(label[5],"SSBL");
    strcpy(label[6],"FSK1");
    strcpy(label[7],"FSK2");
    strcpy(label[8],"BPSK");*/

    /* Open files */
    input1 = fopen(argv[1],"r");
    input2 = fopen(argv[2],"r");
    input3 = fopen(argv[3],"r");
    output1 = fopen(argv[4],"w");
    output2 = fopen(argv[5],"w");

    /* Read in training data */
    printf("Reading in train file ...\n");
    get_data(data,input1,input2);

    /* Close input files */
    fclose(input1);
    fclose(input2);

    /* Compute scores */
    printf("Computing scores ...\n");
    fprintf(output1,"Test Vector: Output Class\n");

    /* Loop through all test vectors, compute nearest neighbours */
    for (i=0;i<TEST;i++)
    {
        /* Read in test vector */
        get_vector(test_vector,input3);

        /* Compute neighbours */
        compute_neighbor(data,test_vector,&i);

        /* Print results to screen */
        fprintf(output1," %d %s \n",i+1,label[i]);
    }
}

```



```

    fprintf(output2, "%d\n", j);
}

/* Save files */
fclose(input3);
fclose(output1);
fclose(output2);
}

/*****
void get_data(data,input1,input2) /* Read in data files, outputs of the
                                RANDCOMBINE.C program */
float data[CLASS][TRAIN][VECTOR]; /* Data array */
FILE *input1,*input2;
{
    int i,j,k,total,max_index;
    int class_count[CLASS]; /* Count the number read in from each class */
    float max_temp;

    /* Total number of training vectors to be read in */
    total=CLASS*TRAIN;

    /* Zero count */
    for (i=0;i<CLASS;i++)
    {
        class_count[i]=0;
    };

    /* Loop through all vectors to be read in */
    for (i=0;i<total;i++)
    {
        max=-1000.00;
        max_index=-1;

        /* Scan line in label file to find the maximum entry, indicating the
           class number to which corresponding line vector in input_file belongs
           to */
        for(k=0;k<CLASS;k++)
        {
            fscanf(input2, "%f", &temp);
            if(temp>max)
            {
                max_index=k;
                max=temp;
            };
        };

        /* Read in training vector to corresponding class portion of data[][] */
        for (k=0;k<VECTOR;k++)
        {
            fscanf(input1, "%f", &temp);
            data[max_index][class_count[max_index]][k]=temp;
        }

        /* Increment class-read count */
        class_count[max_index]++;
    }
}

/*****
void get_vector(test_vector,input) /* Read in test vector from test_file */
float test_vector[VECTOR];
FILE *input;
{
    int i;

    /* Loop through all dimensions and read vector */
    for(i=0;i<VECTOR;i++)
    {
        fscanf(input, "%f", &test_vector[i]);
    }
}

/*****
void compute_neighbor(data,test_vector,class) /* Compute nearest neighbour
                                              of inputted test vector
                                              and output its class */
float data[CLASS][TRAIN][VECTOR], /* Training vectors */

```

```

test_vector[VECTOR]; /* Testing vectors */
int *class; /* Output class */

{
    int i,j,k;
    min_class; /* Minimum distance class */
    index[K]; /* Class numbers of the k nearest neighbours */
    float tmp1,tmp2;
    min[K+1]; /* Array containing minimum distances for K-nearest neighbours */

    /* Initialize index[] to zero, and all min[] entries */
    for (k=0;k<K;k++)
    {
        index[k]=0;
        min[k]=1.0E20;
    };
    min[K]=min[K-1];

    min_class = -1;

    /* Loop through all data and compute and update nearest neighbour information */

    /* Loop through all classes */
    for (i=0;i<CLASS;i++)
    {
        /* Loop through all training vectors for class */
        for (j=0;j<TRAIN;j++)
        {
            tmp1 = 0.0;
            /* Compute Euclidean distance from test vector */
            for (k=0;k<VECTOR;k++)
            {
                tmp2 = data[i][j][k] - test_vector[k];
                tmp1 = tmp1 + (tmp2*tmp2);
            }

            /* If distance less than maximum of K-nearest neighbours, update
            min[] and index[] by sorting, shortest distance in [0] locations,
            largest in [K-1] */
            if(tmp1<min[K])
            {
                sort_min(tmp1,min,index,i);
            }
        }
    }

    /* Return class */
    *class = get_class(index);
}

/*****
void sort_min(valeur, min, index, class) /* Sort routine, compare valeur
to entries in min[], and if less than
some, insert and sort array */

float valeur; /* Distance to be checked */
min[K+1]; /* Array of minimum distances */
int index[K]; /* Classes of corresponding entries in min[] */
class; /* Class of vector with distance valeur */

{
    int ii,jj;

    /* Sort, and if valeur is less than entry in min[], insert and erase
    largest entry. Update index[] appropriately */
    for (ii = 0; ii < (K - 1); ii++) {
        if (valeur < min[ii]) {
            for (jj = (K - 1); jj >= (ii + 1); jj--) {
                min[jj] = min[jj - 1];
                index[jj] = index[jj - 1];
            }
            min[ii] = valeur;
            index[ii] = class;
            min[K] = min[K - 1];
            return;
        }
    }
    min[(K - 1)] = valeur;
    index[(K - 1)] = class;
    min[K] = min[(K - 1)];
}

```

```

/*****
int get_class(index) /* Return class number based on the most dominant class
in the K-nearest neighbours. */

int index[K]; /* Array containing the classes of the K-nearest neighbours */
{
    int counter[CLASS], /* Counts the number of neighbours from each class */
        ii, jj, max, class;

    /* Zero count */
    for (ii = 0; ii < CLASS; ii++)
        counter[ii] = 0;

    /* Count each class */
    for (ii = 0; ii < CLASS; ii++) {
        for (jj = 0; jj < K; jj++) {
            if (ii == index[jj])
                counter[ii]++;
        }
    }

    /* Find maximum class, corresponding to highest count */
    max = -1;
    for (ii = 0; ii < CLASS; ii++) {
        if (counter[ii] > max) {
            max = counter[ii];
            class = ii;
        }
    }

    /* Return value */
    return(class);
}
*****/

```

```

...../
/* Program: NBP2.C */
/* Written by: Nasir Ghani */
/* Purpose: Backpropagation program with single hidden layer. The */
/* program runs in an interactive environment, allowing the */
/* user to specify learning rates, momentum, training files, */
/* and weight files. The program requires that the input */
/* files be multiples of PPS (default=90, same as WARP), */
/* and this is the batch size used to update the weights. */
/* The weights file is read and written in the following */
/* form: */
/* weight_from_first_hidden_to_first_input */
/* weight_from_first_hidden_to_second_input */
/* . */
/* weight_from_first_hidden_to_last_input */
/* bias_of_first_hidden */
/* weight_from_second_hidden_to_first_input */
/* weight_from_second_hidden_to_second_input */
/* . */
/* weight_from_second_hidden_to_last_input */
/* bias_of_second_hidden */
/* . */
/* weight_from_last_output_to_last_hidden */
/* bias_of_last_output */
/* number_of_weights */
/*
For starting off training, it is assumed that an input */
/* random file of weights is already generated, with the */
/* values typically ranging from -0.03 to +0.03. */
/* The input and output files are just the training files, */
/* with the input file containing one training vector per */
/* and the output file containing the corresponding desired */
/* vector for it. */
/* To use the program, the network has to be specified in the */
/* constants.h file and the program compiled. This file has */
/* the following format: */
/*
#define NIU <number_input_neurons> */
/* #define NHU <number_hidden_neurons> */
/* #define NOU <number_output_neurons> */
/* #define NUM_SETS <number_training_sets> */
/* #define WEIGHT_FILE <weight_file_to_start> */
/* #define INPUT_PATTERN_FILE <input_train_file> */
/* #define OUTPUT_PATTERN_FILE <desired_file> */
/*
Note that the INPUT_PATTERN_FILE and OUTPUT_PATTERN_FILE */
/* have to both be of length NUM_SET*PPS (ie NUM_SETS groups */
/* each of length PPS, the epoch size. */
/* The parser for the */
/* interactive command shell requires the file cparser2.h. */
/*
/* Runline: */
/* bp2 -l <lrc1> <lrc2> -m <momentum> -i <input_file> */
/* -o <output_file> -w <weight_file> */
/*
/* where lrc1 and lrc2 are the learning rates for the input */
/* to hidden and hidden to output layers respectively.
...../
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<strings.h>
#include<memory.h>

#include "constants.h"
#include "cparser2.h"

/* Define command table lookup */
CMDTBL cmdtbl[] =
{
    {read_weights,"read_weight_file"},
    {write_weights,"write_weight_file"},
    {goiter,"go"},
    {quit,"quit"},
}

```

```

        {lrc1, "lrc1="},
        {lrc2, "lrc2="},
        {momentumc, "momentum="},
        {epochnum, "epoch="},
    };

#define PPS 90 /* Patterns Per Set = Epoch Size */
#define HUGE_VALUE 1.0 /* Upper limit for learning rates and momentum */
#define MAX_INT 100000000 /* Upper limit on iteration number */
#define NOCHAR (-2)
#define SAVE_FILE_INIT "bpfull2.wt" /* Default weight filename */

float i_h[NHU][NIU+1][2]; /* i_h[][0] is for input to hidden weights
                             (bias also), i_h[][1] is for momentum of the
                             corresponding weight */
float h_o[NOU][NHU+1][2]; /* h_o[][0] is for hidden to output weights
                             (bias also), h_o[][1] is for momentum of the
                             corresponding weight */
float inputs[NUM_SETS][PPS][NIU+1]; /* Array holding input training vectors */
float outputs[NUM_SETS][PPS][NOU]; /* Array holding desired output vectors */
float h_act[PPS][NHU+1]; /* Matrix to hold hidden layer outputs
                           for a batch size */
float out_act[PPS][NOU]; /* Matrix to hold output layer outputs for
                           a batch size */
float hidden_delta[PPS][NHU+1]; /* Stores intermediate results for speeding
                                  up backpropagation algorithm */
float learning[3]; /* Vector for learning rates, and momentum
                   learning[0]=lrc1, learning[1]=lrc2, learning[2]=momentum */
float error_scale; /* To scale error output over batch */
int set_set_count, epoch_count; /* Counting variables */
float total_error; /* Sums error over iteration, ie. NUM_SETS batches */
char input_pattern_file[256], output_pattern_file[256], weight_file[256],
      saved_weight_file[256], save_file[256]; /* Filename variables */
float lrc1, lrc2, momentum; /* Training constants */
int epoch; /* Iteration Number */
int cmdtblsize; /* Number of shell commands */
int cmdargc; /* Max number of arguments over all shell commands */

void read_weight_file();
void print_nn_constants();
int read_initial_weights();
int read_patterns();
void str_arg();
void write_weight_file();
void go();
int read_comlin();
int initialize();
void exit_program();
void i_h_pass();
void h_o_pass();
void calc_error();
void o_h_pass();
void h_i_pass();
int cmdsearch();

main(argc, argv)
int argc;
char *argv[];
{
    int i, k, templ;
    float temp2;
    char strng[129], rest[129];

    cmdtblsize = 8; /* Number of commands in cparser2.h */
    cmdargc = 1;

    /* Scale error according to whole training set */
    error_scale = NUM_SETS * PPS * NOU;

    /* Check if command line is ok, else exit */
    if (read_comlin(argc, argv))
        exit(0);

    /* Print out header to screen, summarizing input files, weight files, and
       network size */
    print_nn_constants();

    /* If cannot read input and desired training files and weight file, exit */

```

```

if(initialize())
    exit(-1);

/* Loop until exit, process user shell input */
i=1;
while (i==1)
{
    printf("NBP2> "); /* Prompt */
    gets(strg);
    k=cmdsearch(strg,rest);

    /* Check if valid command number */
    if( (k>0) && (k<cmdtblsize) )
    {
        if(k==1)
            read_weight_file(rest);

        if(k==2)
            write_weight_file(rest);

        if(k==3)
            go(rest);

        if(k==4)
            exit_program(rest);

        if(k==5)
        {
            if(rest!=NULL && strlen(rest)>0)
            {
                if(sscanf(rest,"%f",&temp2)==1)
                {
                    /* Limit value under 1 */
                    if(temp2<HUGE_VALUE)
                    {
                        lrc1=temp2;
                    }
                    else
                    {
                        lrc1=HUGE_VALUE;
                    };
                };
                printf("lrc1=%f\n",lrc1);
            };
        };

        if(k==6)
        {
            if(rest!=NULL && strlen(rest)>0)
            {
                if(sscanf(rest,"%f",&temp2)==1)
                {
                    /* Limit value under 1 */
                    if(temp2<HUGE_VALUE)
                    {
                        lrc2=temp2;
                    }
                    else
                    {
                        lrc2=HUGE_VALUE;
                    };
                };
                printf("lrc2=%f\n",lrc2);
            };
        };

        if(k==7)
        {
            if(rest!=NULL && strlen(rest)>0)
            {
                if(sscanf(rest,"%f",&temp2)==1)
                {
                    /* Limit value under 1 */
                    if(temp2<HUGE_VALUE)
                    {
                        momentum=temp2;
                    }
                    else
                    {

```

```

        momentum=HUGE_VALUE;
    };
    printf("momentum=%f\n",momentum);
};

if(k==8)
{
    if((rest!=NULL && strlen(rest)>0)
    {
        if(sscanf(rest,"%d",&temp1)==1)
        {
            epoch=temp1;
        };
        printf("Epoch = %d\n",epoch);
    };
};

}
else
{
    printf("Invalid Entry\n");
};

};

}

/*****
void read_weight_file(arglist) /* Read weight file */
char *arglist;
{
    char filename[256]; /* Weight file name */
    FILE *fd;
    int epoch_number;
    int iu, hu, ou;

    /* Open file, if not there, exit */
    str_arg(arglist, "weights filename: ", filename);
    if ((fd = fopen(filename, "r")) == NULL) {
        fprintf(stderr, "Cannot open %s\n", filename);
        return;
    }

    /* Read weights for each layer */
    printf("Reading weights from %s ... ", filename);
    for(hu=0;hu<NHU;hu++){
        for(iu=0;iu<(NIU+1);iu++){
            fscanf(fd,"%f",&i_h[hu][iu][0]);
            i_h[hu][iu][1] = 0.0;
        }

        for(ou=0;ou<NOU;ou++){
            for(hu=0;hu<NHU+1;hu++){
                fscanf(fd,"%f",&h_o[ou][hu][0]);
                h_o[ou][hu][1] = 0.0;
            }

            fscanf(fd,"%d epochs\n",&epoch_number);
            epoch = epoch_number;

            printf("Read\n");
            fclose(fd);
        }

    }

/*****
int read_patterns() /* Read input files and desired output file */
{
    int iu, hu, ou;
    int set, p;
    FILE *fp;

    /* Open input file, if not, exit */
    if((fp=fopen(input_pattern_file, "r")) == NULL)
    {
        printf("Cannot open %s\n",input_pattern_file);
        return(-1);
    };

    /* Read inputs into INPUT[] */

```



```

printf("Reading input patterns...");
for (set=0; set<NUM_SETS; set++)
{
    for(p=0; p<PPS; p++)
    {
        inputs[set][p][0]=1.0; /* For bias */
        for(iu=0; iu<NIU; iu++)
        {
            fscanf(fp, "%f", &inputs[set][p][iu+1]);
        };
    };
    fclose(fp);
    printf("Read\n");

    /* Open desired file, if not, exit */
    if(((fp=fopen(output_pattern_file, "r")) == NULL))
    {
        printf("Cannot open %s\n", output_pattern_file);
        return(-1);
    };

    /* Read desired outputs into OUTPUTS[] */
    printf("Reading output patterns...");
    for (set=0; set<NUM_SETS; set++)
    {
        for(p=0; p<PPS; p++)
        {
            for(ou=0; ou<NOU; ou++)
            {
                fscanf(fp, "%f", &outputs[set][p][ou]);
            };
        };
    };
    printf("Read\n");

    fclose(fp);
};

/*****
void i_h_pass(inputs, index, i_h_act) /* Forward pass, from input to hidden,
                                   compute hidden outputs to input batch */

float inputs[NUM_SETS][PPS][NIU+1]; /* Input array */
int index; /* Index points to the particular batch
           size in the INPUTS[][] array
           (ie. NUM_SETS index INPUTS[][] array */
float i_h[NHU][NIU+1][2]; /* Input to hidden weights */
float h_act[PPS][NHU+1]; /* Outputs of routine stored in this
                          hidden activation array */
{
    int i,j,k;
    float sum;

    /* Loop over all batch size */
    for (i=0; i<PPS; i++)
    {
        /* Loop over all hidden neurons */
        for (j=0; j<NHU; j++)
        {
            sum=0.0;
            /* Compute weighted sum for each hidden neuron */
            for (k=0; k<NIU; k++)
            {
                sum=sum+(inputs[index][i][k]*i_h[j][k][0]);
            };
            /* Pass sum through sigmoid nonlinearity */
            h_act[i][j]=1.0/(1.0+exp(-1.0*sum));
        };
        /* Set (NHU+1)-th output....ie for bias to output neuron in next stage */
        h_act[i][NHU]=1.0;
    };
};

/****
void h_o_pass(h_act, h_o, out_act) /* Forward pass, from hidden to output,
                                   compute outputs from hidden outputs */

```

```

float h_act[PPS][NHU+1]; /* Hidden outputs for the batch */
float h_o[NOU][NHU+1][2]; /* Hidden to output weights */
float out_act[PPS][NOU]; /* Outputs corresponding to inputs who have
                           hidden outputs in H_ACT[] */
{
    int i,j,k;
    float sum;

    /* Loop over all batch entries */
    for (i=0;i<PPS;i++)
    {
        /* Loop over all outputs */
        for (j=0;j<NOU;j++)
        {
            sum=0.0;
            /* Computed weighted sum of hidden activations */
            for (k=0;k<=(NHU);k++)
            {
                sum=sum+(h_act[i][k]*h_o[j][k][0]);
            };
            /* Pass sum through sigmoid nonlinearity */
            out_act[i][j]=1.0/(1.0+exp(-1.0*sum));
        };
    };
};

/*=====*/
void calc_error(outputs,index,out_act,total_error) /* Compute error for batch */
{
    float outputs[NUM_SETS][PPS][NOU]; /* Desired outputs for the training data */
    int index; /* Index to current
               batch in INPUTS[] and OUTPUTS[] array */
    float out_act[PPS][NOU]; /* Actual output for the network for input batch */
    float *total_error; /* Total error squared over all input file, ie. over all
                        NUM_SETS batches */
    {
        int i,j,k;
        float sum;

        /* Loop over all batch entries */
        for (i=0;i<PPS;i++)
        {
            /* Loop through all outputs */
            for (j=0;j<NOU;j++)
            {
                /* Compute error */
                *total_error += (outputs[index][i][j]-out_act[i][j]) *
                    (outputs[index][i][j]-out_act[i][j]);
            };
        };
    };
};

/*=====*/

void o_h_pass(outputs,index,h_act,out_act,h_o,learning,hidden_delta)
/* Backward pass from output to hidden layers, compute derivatives pertaining
   to the weights and update momentum in h_o[][][1] */
{
    float outputs[NUM_SETS][PPS][NOU]; /* Desired outputs array */
    int index; /* Index to current batch */
    float h_act[PPS][NHU+1]; /* Hidden activations */
    float out_act[PPS][NOU]; /* Outputs of network for batch */
    float h_o[NOU][NHU+1][2]; /* Hidden to output weights */
    float learning[3]; /* Learning rates and momentum */
    float hidden_delta[PPS][NHU+1]; /* Temporary variable to store intermediate
                                     result to backpropagate error to update weights
                                     for input to hidden layer weights */
    {
        int i,j,k;
        float sum;
        float output_delta[PPS][NOU]; /* Temporary variable, accounting for error and
                                     corresponding derivative, for each output
                                     unit, used to compute derivatives */
    }
}

```

```

/* Zero values to be backpropagated */
for (i=0;i<PPS;i++)
{
    for (j=0;j<NHU;j++)
    {
        hidden_delta[i][j]=0.0;
    };
};

/* Loop over all batch entries and outputs and compute error-related term */
for (i=0;i<PPS;i++)
{
    for (j=0;j<NOU;j++)
    {
        output_delta[i][j]=(outputs[index][i][j]-out_act[i][j])*
            (out_act[i][j]*(1.0-out_act[i][j]));
    };
};

/* Loop over all output neurons */
for (i=0;i<NOU;i++)
{
    /* Loop over all hidden neurons connecting to the output neuron */
    for (j=0;j<NHU;j++)
    {
        /* Zero sum */
        sum=0.0;
        /* Loop over all entries in training batch */
        for (k=0;k<PPS;k++)
        {
            /* Update change */
            sum=sum+(output_delta[k][i]*h_act[k][j]);
            /* Update quantity to be backpropagated for input-hidden layer weights */
            hidden_delta[k][j] += (output_delta[k][i]*h_o[i][j][0]);
        };

        /* Update weights according to change over all training */
        h_o[i][j][0] += h_o[i][j][1] + learning[1]*sum;
        /* Update momentum related term */
        h_o[i][j][1] = learning[2]*(h_o[i][j][1]+sum);
    };
};

/* ..... */

void h_i_pass(inputs,index_hidden_delta,h_act,out_act,learning,i_h)
/* Backward pass from hidden to input layers, compute derivatives pertaining
   to the weights and store momentum update in i_h[][][1] */

float inputs[NUM_SETS][PPS][NIU+1]; /* Input training vectors */
int index; /* Index to current batch */
float hidden_delta[PPS][NHU+1]; /* Backpropagated error quantity from
    hidden-output layer */
float h_act[PPS][NHU+1]; /* Hidden activation for current batch */
float i_h[NHU][NIU+1][2]; /* Weights from input to hidden layer */
float learning[3]; /* Learning rates and momentum */

{
    int i,j,k;
    float sum;
    float h_delta[PPS][NHU];

    /* Using backpropagated quantity, scale with derivative to generate another
       quantity to be used in generating derivative, corresponding to each hidden
       neuron */

    for (i=0;i<PPS;i++)
    {
        for (j=0;j<NHU;j++)
        {
            h_delta[i][j]=(hidden_delta[i][j])*(h_act[i][j])*(1.0-h_act[i][j]);
        };
    };

    /* Loop over all hidden neurons */
    for (i=0;i<NHU;i++)

```

```

{
/* Loop over all input neurons */
for (j=0;j<NIU;j++)
{
/* Zero change */
sum=0.0;
for (k=0;k<PPS;k++)
{
/* Update change */
sum=sum+(h_delta[k][i]*inputs[index][k][j]);
};
/* Update weight */
i_h[i][j][0] += i_h[i][j][1] + learning[0]*sum;
/* Update momentum term */
i_h[i][j][1] = learning[2]*(i_h[i][j][1]+sum);
};
};
};

/*****
void go(arglist) /* Routine to start backpropagation routine, in response
to user typing "go <n>" at NBP2 prompt. Will go n
iterations else if n not specified, will loop until
stopped by user */
char *arglist;
{
char c;
int counting_epochs=0;
int run_epoch_count=0;

/* Check if number of iterations specified (n) */
if (arglist==NULL && strlen(arglist)>0)
if (scanf(arglist,"%d",&run_epoch_count)==1)
counting_epochs=1;
if (counting_epochs==1 & run_epoch_count<=0)
return;

/* Start Iteration-Error display */
printf(" Epoch      Average Error per Output Unit\n");
while(1) {
learning[0]=lrc1;
learning[1]=lrc2;
learning[2]=momentum;

/* Zero iteration error */
total_error=0.0;

/* Loop over all batch of size PPS */
for (set=0;set<NUM_SETS;set++)
{
/* Input to hidden pass for set batch */
i_h_pass(inputs,set,i_h_act);
/* Hidden to output pass */
h_o_pass(h_act,h_o_act);
/* Calculate error for batch */
calc_error(outputs,set,out_act,&total_error);
/* Backward pass from output to hidden, update weights in hidden-output layer*/
o_h_pass(outputs,set,h_act,out_act,h_o_learning,hidden_delta);
/* Backward pass from hidden to input, update weights in input-hidden layer */
h_i_pass(inputs,set,hidden_delta,h_act,out_act,learning,i_h);
};

/* Increment iteration count */
epoch_count=epoch_count+1;

/* Print error */
total_error/= error_scale;
printf("%d      %f\n",epoch_count,total_error);

/* Check if done n iterations */
if(counting_epochs)
{
run_epoch_count -= 1;
if(run_epoch_count<=0)
{
return;
};
};
};

```

```

    };
};
/*****
int read_comlin(argc, argv) /* Read command line input, and parse */
{
    int argc;
    char *argv[];
    {
        double atof();
        int i;

        /* Load default values, overwrite later with user specified values if given */
        strcpy(input_pattern_file, INPUT_PATTERN_FILE);
        strcpy(output_pattern_file, OUTPUT_PATTERN_FILE);
        strcpy(weight_file, WEIGHT_FILE);
        strcpy(saved_weight_file, SAVE_FILE_INIT);
        lrc1 = 0.1;
        lrc2 = 0.1;
        momentum = 0.5;

        /* Parse input and overwrite corresponding default settings */
        for(i=1; i<argc; i++)
            if(argv[i][0]!='-')
                switch(argv[i][1]){
                    case 'i':
                        i++;
                        strcpy(input_pattern_file, argv[i]);
                        break;
                    case 'o':
                        i++;
                        strcpy(output_pattern_file, argv[i]);
                        break;
                    case 'w':
                        i++;
                        strcpy(weight_file, argv[i]);
                        break;
                    case 's':
                        i++;
                        strcpy(saved_weight_file, argv[i]);
                        break;
                    case 'l':
                        lrc1 = atof(argv[++i]);
                        lrc2 = atof(argv[++i]);
                        break;
                    case 'm':
                        momentum = atof(argv[++i]);
                        break;
                    default:
                        printf("Usage: host [-i learn1 learn2] [-m momentum]");
                        printf(" [-i input_patn_file] [-o output_patn_file] ");
                        printf("[-w initial_weight_file] [-s saved_weight_file]\n");
                        return(-1);
                }
        return(0);
    }
}
/*****
int initialize () /* Read weights, read input and desired training files */
{
    if(read_initial_weights())
        return(-1);
    if(read_patterns())
        return(-1);
    return(0);
}

/*****
void write_weight_file(arglist) /* Saves weight file */
{
    char *arglist;
    {
        char filename[256];
        FILE *fd;
        int iu, hu, ou;

        /* Open file to write to, if not possible, exit */
        str_arg(arglist, "weights filename: ", filename);
        if ((fd = fopen(filename, "w")) == NULL) {
            fprintf(stderr, "could not open %s\n", filename);
            return;
        }
    }
}

```

```

/* Loop through each layer, input to hidden first and hidden to output and
write out...see header at start of program for format */
for(hu=0;hu<NHU;hu++){
    for(iu=0;iu<NIU+1;iu++){
        fprintf(fd, "%f\n", i_h[hu][iu][0]);
    }

    for(ou=0;ou<NOU;ou++){
        for(hu=0;hu<NHU+1;hu++){
            fprintf(fd, "%f\n", h_o[ou][hu][0]);
        }
    }

/* Write out number of iterations */
    fprintf(fd, "%d epochs\n", epoch);
    printf("written\n");
    fclose(fd);
}
/*****/
int read_initial_weights() /* Reads initial weights */
{
    int iu, hu, ou;
    FILE *fp;

/* Open initial weights file */
    if((fp = fopen(weights_file, "r")) == NULL){
        printf("cannot open %s\n", weights_file);
        return(-1);
    }

/* Read in weights, format specified in header */
    printf("Reading weights ... ");
    for(hu=0; hu<NHU; hu++){
        for(iu=0; iu<(NIU+1); iu++){
            fscanf(fp, "%f", &i_h[hu][iu][0]);
            i_h[hu][iu][1] = 0.0;
        };

        for(ou=0; ou<NOU; ou++){
            for(hu=0;hu<NHU+1;hu++){
                fscanf(fp, "%f", &h_o[ou][hu][0]);
                h_o[ou][hu][1] = 0.0;
            };
        }

/* Read in number of iterations done with these weights */
        fscanf(fp, "%d epochs\n", &epoch);
        fclose(fp);
        printf("read\n");
        return(0);
    }
/*****/
void print_nn_constants() /* Print header information to screen, summarizing
input and details of network */
{
    printf("Starting fully connected NN with 1 hidden layer\n");
    printf("%3d Input units (w/o bias unit)\n", NIU);
    printf("%3d Hidden units (w/o bias unit)\n", NHU);
    printf("%3d Output units\n", NOU);
    printf("\nLearning Rates %6.3f %6.3f\n", lrc1, lrc2);
    printf("Momentum %6.3f\n", momentum);
    printf("\nTraining Files\n");
    printf("Input Pattern %s\n", input_pattern_file);
    printf("Output Pattern %s\n", output_pattern_file);
    printf("Weight %s\n", weight_file);
}
/*****/
void str_arg(string, prompt, name) /* String to argument conversion */
char *string, *prompt, *name;
{
    if (string==NULL || sscanf(string, "%s", name)!=1) {
        printf(prompt);
        scanf("%s", name);
    };
}
/*****/
void exit_program(arglist) /* In response to "quit" at NBP2 prompt */
char *arglist;
{
    char answer[12];

```

```

/* Check if weights to be saved, if so, save them and exit */
str_arg(arglist, "save weights file [y/n]: ", answer);
if(answer[0]!='n' && answer[0]!='N')
    write_weight_file(save_file);
exit(0);
}
/*****
int cmdsearch( str, rest ) /* Search for command number given input string
                           entered by user. Return arguments in rest */
char      *str; /* Input string */
char      *rest; /* Arguments */
{
    int      i, l;

/* Loop over all command entries */
    for (i=0; i<cmdtblsize; i++)
    {
        l = strlen( cmdtbl[i].cmdstr );
        /* If match found, return number, and arguments */
        if ( !memcmp( str, cmdtbl[i].cmdstr, l ) )
        {
            strcpy( rest, str + l );
            return (cmdtbl[i].cmdno);
        }
    }
    strcpy( rest, str );
    return (CMD_NULL);
}
*****/

```



```
/*  
  Constants.h  
*/
```

```
#define NIU 18  
#define NHU 25  
#define NOU 2  
#define NUM_SETS 360  
#define WEIGHT_FILE "bpfull3.wt"  
#define INPUT_PATTERN_FILE "/home/neuron/ghani/PER/P256/SHORT_div_10.IN"  
#define OUTPUT_PATTERN_FILE "/home/neuron/ghani/PER/P256/TRAIN.OUT"
```

```

/*
 *      file:      cparser2.h
 *      desc:      simple command parser
 *      by:        patrick ko
 *      date:      22 aug 91
 */

#define      CMD_NULL 0

typedef      struct      {
    int      cmdno;
    char      *cmdstr;
}      CMDTBL;

/*
 *      #define your own commands here starting from 1
 *      *-modifiable*-
 */

#define      read_weights      1
#define      write_weights      2
#define      goiter      3
#define      quit      4
#define      lra      5
#define      lrb      6
#define      momentumc      7
#define      epochnum      8

```

```

/*****
/* Program: BP2OBD.C */
/* Written by: Nasir Ghani */
/* Purpose: Optimal Brain Damage (OBD) with single hidden layer. The */
/* program runs in an interactive environment, allowing the */
/* user to specify learning rates, momentum, training files, */
/* weight files, and deletion thresholds, and ratios. */
/* The program requires that the input */
/* files be multiples of PPS (default=90, same as WARP), */
/* and this is the batch size used to update the weights, */
/* and compute double derivatives for the saliencies. */
/* The weight and saliency files have the following format: */
/*
/* weight_from_first_hidden_to_first_input */
/* weight_from_first_hidden_to_second_input */
/*
/* weight_from_first_hidden_to_last_input */
/* bias_of_first_hidden */
/* weight_from_second_hidden_to_first_input */
/* weight_from_second_hidden_to_second_input */
/*
/* weight_from_second_hidden_to_last_input */
/* bias_of_second_hidden */
/*
/* weight_from_last_output_to_last_hidden */
/* bias_of_last_output */
/* number_of_weights */
/*
/* For starting off training, it is assumed that an input */
/* random file of weights is already generated, with the */
/* values typically ranging from -0.03 to +0.03. */
/* The input and output files are just the training files, */
/* with the input file containing one training vector per */
/* and the output file containing the corresponding desired */
/* vector for it. */
/* To use the program, the network has to be specified in the */
/* constants.h file and the program compiled. This file has */
/* the following format:
/*
/* #define NIU <number_input_neurons> */
/* #define NHU <number_hidden_neurons> */
/* #define NOU <number_output_neurons> */
/* #define NUM_SETS <number_training_sets> */
/* #define WEIGHT_FILE <weight_file_to_start> */
/* #define INPUT_PATTERN_FILE <input_train_file> */
/* #define OUTPUT_PATTERN_FILE <desired_file> */
/*
/* Note that the INPUT_PATTERN_FILE and OUTPUT_PATTERN_FILE */
/* have to both be of length NUM_SET*PPS (ie NUM_SETS groups */
/* each of length PPS, the epoch size. (b&v) */
/* The parser for the interactive command shell requires the */
/* file cparser2obd.h.
/*
/* Runline:
/* nbp2obd -l <lrc1> <lrc2> -m <momentum> -i <input_file> */
/* -o <output_file> -w <weight_file>
/*
/* where lrc1 and lrc2 are the learning rates for the input */
/* to hidden and hidden to output layers respectively.
*****/
#include<math.h>
#include<stdio.h>
#include<stdlib.h>

#include<string.h>
#include<strings.h>

/* In my directory */
#include "constants.h"
#include "cparser2obd.h"

CMDTBL cmdtbl[] =
{
    {read_weights,"read_weights_file"},
    {write_weights,"write_weight_file"},
    {write_dd,"write_dd_file"},
    {write_saliency,"write_saliency_file"},
    {obdmain,"obd"},
}

```

```

        {goiter,"go"},
        {quit,"quit"},
        {lrca, "lrca1="},
        {lrcb, "lrc2="},
        {momentumc, "momentum="},
        {threshold, "thresh="},
        {decayconstant, "decay="},
        {epochnum, "epoch="},
    };

#define FPS 90 /* Patterns Per Set = Epoch Size */
#define HUGE_VALUE 1.0 /* Upper limit for learning rates and
    momentum */
#define MAX_INT 100000000 /* Upper limit on iteration number */
#define SAVE_FILE_INTT "bpfull2.wt" /* Default weight file */
#define NOCHAR (-2)

float i_h[NHU+1][3]; /* i_h[][0] is for input to hidden weights
    (bias also), i_h[][1] is for momentum of the
    corresponding weight */
float h_o[NOU][3]; /* h_o[][0] is for hidden to output weights
    (bias also), h_o[][1] is for momentum of the
    corresponding weight */
float inputs[NUM_SETS][FPS][NHU+1]; /* Array holding input training vectors */
float outputs[NUM_SETS][FPS][NOU]; /* Array holding desired output vectors */
float h_act[FPS][NHU+1]; /* Matrix to hold hidden layer outputs
    for a batch size */
float out_act[FPS][NOU]; /* Matrix to hold output layer outputs for
    a batch size */
float hidden_delta[FPS][NHU+1]; /* Stores intermediate results for speeding
    up backpropagation algorithm */
float hidden_ddelta[FPS][NHU+1]; /* Stores intermediate results for speeding
    up OBD calculations */
float learning[3]; /* Vector for learning rates, and momentum
    learning[0]=lr1, learning[1]=lr2, learning[2]=momentum */
float error_scale; /* To scale error output over batch */
int set_set_count, epoch_count; /* Counting variables */
float total_error; /* Sums error over iteration, ie. NUM_SETS batches */
char input_pattern_file[256], output_pattern_file[256], weight_file[256],
    saved_weight_file[256], save_file[256]; /* Filename variables */
float lr1, lr2, momentum, thresh, decay; /* Training and pruning parameters */
int epoch; /* Iteration Number */
int cmdtblsize; /* Number of shell commands */
int cmdargc; /* Max number of arguments over all shell commands */

void read_weight_file();
void print_nn_constants();
int read_initial_weights();
int read_patterns();
void str_arg();
void write_weight_file();
void write_dd_file();
void write_saliency_file();
void go();
void obd();
int read_conlin();
int initialize();
void exit_program();
void i_h_pass();
void h_o_pass();
void calc_error();
void o_h_pass();
void h_i_pass();

main(argc,argv)
int argc;
char *argv[];

{
    int i,k,temp1;
    float temp2;
    char strng[129], res[129];

    cmdtblsize=13; /* Number of commands in cparser2obd.h */
    cmdargc=1;

    /* Scale error according to whole training set */
    error_scale=NUM_SETS*FPS*NOU;

```

```

/* Check if command line is ok, else exit */
if(read_comlin(argc,argv))
    exit(0);

/* Print out header to screen, summarizing input files, weight files, and
network size */
print_nn_constants();

/* If cannot read input and desired training files and weight file, exit */
if(initialize())
    exit(-1);

/* Loop until exit, process user shell input */
i=1;
while (i==1)
{
    printf("NBP2OBD> "); /* Prompt */
    gets(stng);
    k=cmdsearch(stng,rest);

    /* Check if valid command number */
    if( (k>0) && (k<cmdtblsize) )
    {
        if(k==1)
            read_weight_file(rest);

        if(k==2)
            write_weight_file(rest);

        if(k==3)
            write_dd_file(rest);

        if(k==4)
            write_saliency_file(rest);

        if(k==5)
            obd();

        if(k==6)
            go(rest);

        if(k==7)
            exit_program(rest);

        if(k==8)
        {
            if(rest!=NULL && strlen(rest)>0)
            {
                if(sscanf(rest,"%f",&temp2)==1)
                {
                    /* Limit value under 1 */
                    if(temp2<HUGE_VALUE)
                    {
                        lrc1=temp2;
                    }
                    else
                    {
                        lrc1=HUGE_VALUE;
                    }
                }
                printf("lrc1=%f\n",lrc1);
            }
        }

        if(k==9)
        {
            if(rest!=NULL && strlen(rest)>0)
            {
                if(sscanf(rest,"%f",&temp2)==1)
                {
                    /* Limit value under 1 */
                    if(temp2<HUGE_VALUE)
                    {
                        lrc2=temp2;
                    }
                    else
                    {
                        lrc2=HUGE_VALUE;
                    }
                }
            }
        }
    }
}

```

```

    printf("lrc2=%f\n",lrc2);
};
};

if(k==10)
{
    if(rest!=NULL && strlen(rest)>0)
    {
        if(sscanf(rest,"%f",&temp2)==1)
        {
            /* Limit value under 1 */
            if(temp2<HUGE_VALUE)
            {
                momentum=temp2;
            }
            else
            {
                momentum=HUGE_VALUE;
            }
        }
        printf("momentum=%f\n",momentum);
    };
};

if(k==11)
{
    if(rest!=NULL && strlen(rest)>0)
    {
        if(sscanf(rest,"%f",&temp2)==1)
        {
            thresh=temp2;
        }
        printf("thresh=%f\n",thresh);
    };
};

if(k==12)
{
    if(rest!=NULL && strlen(rest)>0)
    {
        if(sscanf(rest,"%f",&temp2)==1)
        {
            decay=temp2;
        }
        printf("decay=%f\n",decay);
    };
};

if(k==13)
{
    if(rest!=NULL && strlen(rest)>0)
    {
        if(sscanf(rest,"%d",&temp1)==1)
        {
            epoch=temp1;
        }
        printf("Epoch = %d\n",epoch);
    };
};

}
else
{
    printf("Invalid Entry\n");
};

};

}

/*****
void read_weight_file(arglist) /* Read weight file */
char *arglist;
{
    char filename[256]; /* Weight file name */
    FILE *fd;
    int epoch_number;
    int in, hu, ou;

    /* Open file, if not there, exit */

```

```

str_arg(arglist, "weights filename: ", filename);
if ((fd = fopen(filename, "r")) == NULL) {
    fprintf(stderr, "Cannot open %s\n", filename);
    return;
}

```

```

/* Read weights for each layer */
printf("Reading weights from %s ... ", filename);
for(hu=0; hu<NHU; hu++){
    for(iu=0; iu<(NIU+1); iu++){
        fscanf(fd, "%f", &i_h[hu][iu][0]);
        i_h[hu][iu][1] = 0.0;
        i_h[hu][iu][2] = 0.0;
    }
}

```

```

for(ou=0; ou<NOU; ou++){
    for(hu=0; hu<NHU+1; hu++){
        fscanf(fd, "%f", &h_o[ou][hu][0]);
        h_o[ou][hu][1] = 0.0;
        h_o[ou][hu][2] = 0.0;
    }
}

```

```

fscanf(fd, "%d epoch\n", &epoch_number);
epoch = epoch_number;

```

```

printf("Read\n");
fclose(fd);
}

```

```

/*****
int read_patterns() /* Read input files and desired output file */

```

```

{
    int iu, hu, ou;
    int set, p;
    FILE *fp;

```

```

/* Open input file, if not, exit */
if(((fp=fopen(input_pattern_file, "r")) == NULL)
{
    printf("Cannot open %s\n", input_pattern_file);
    return(-1);
};

```

```

/* Read inputs into INPUT[] */
printf("Reading input patterns...\n");

```

```

for (set=0; set<NUM_SETS; set++)
{
    for(p=0; p<PPS; p++)
    {
        inputs[set][p][0]=1.0; /* For bias */
        for(iu=0; iu<NIU; iu++)
        {
            fscanf(fp, "%f", &inputs[set][p][iu+1]);
        };
    };
};
fclose(fp);

```

```

/* Open desired file, if not, exit */
if(((fp=fopen(output_pattern_file, "r")) == NULL)
{
    printf("Cannot open %s\n", output_pattern_file);
    return(-1);
};

```

```

/* Read desired outputs into OUTPUTS[] */
printf("Reading output patterns...\n");
for (set=0; set<NUM_SETS; set++)

```

```

{
    for(p=0; p<PPS; p++)
    {
        for(ou=0; ou<NOU; ou++)
        {
            fscanf(fp, "%f", &outputs[set][p][ou]);
        };
    };
};

```

```

fclose(fp);

```



```

};

/*****
void i_h_pass(inputs,index_i,h_act) /* Forward pass, from input to hidden,
compute hidden outputs to input batch */

float inputs[NUM_SETS][PPS][NIU+1]; /* Input array */
int index; /* Index points to the particular batch
size in the INPUTS[][][] array
(ie. NUM_SETS index INPUTS[][][] array */
float i_h[NHU][NIU+1][3]; /* Input to hidden weights */
float h_act[PPS][NHU+1]; /* Outputs of routine stored in this
hidden activation array */
{
int i,j,k;
float sum;

/* Loop over all batch size */
for (i=0;i<PPS;i++)
{
/* Loop over all hidden neurons */
for (j=0;j<NHU;j++)
{
sum=0.0;
/* Compute weighted sum for each hidden neuron */
for (k=0;k<=(NIU);k++)
{
sum=sum+(inputs[index][i][k]*i_h[j][k][0]);
};
/* Pass sum through sigmoid nonlinearity */
h_act[i][j]=1.0/(1.0+exp(-1.0*sum));
};
/* Set (NHU+1)-th output....ie for bias to output neuron in next stage */
h_act[i][NHU]=1.0;
};
};

/*****
void h_o_pass(h_act,h_o,out_act) /* Forward pass, from hidden to output,
compute outputs from hidden outputs */

float h_act[PPS][NHU+1]; /* Hidden outputs for the batch */
float h_o[NOU][NHU+1][3]; /* Hidden to output weights */
float out_act[PPS][NOU]; /* Outputs corresponding to inputs who have
hidden outputs in H_ACT[][] */
{
int i,j,k;
float sum;

/* Loop over all batch entries */
for (i=0;i<PPS;i++)
{
/* Loop over all outputs */
for (j=0;j<NOU;j++)
{
sum=0.0;
/* Computed weighted sum of hidden activations */
for (k=0;k<=(NHU);k++)
{
sum=sum+(h_act[i][k]*h_o[j][k][0]);
};
/* Pass sum through sigmoid nonlinearity */
out_act[i][j]=1.0/(1.0+exp(-1.0*sum));
};
};
};

/*****
void calc_error(outputs,index,out_act,total_error) /* Compute error for batch */

float outputs[NUM_SETS][PPS][NOU]; /* Desired outputs for the training data */
int index; /* Index to current
batch in INPUTS[][][] and OUTPUTS[][][] array */
float out_act[PPS][NOU]; /* Actual output for the network for input batch */
float *total_error; /* Total error squared over all input file, ie. over all
NUM_SETS batches */

```

```

int i,j,k;
float sum;

/* Loop over all batch entries */
for (i=0;i<PPS;i++)
{
    /* Loop through all outputs */
    for (j=0;j<NOU;j++)
    {
        /* Compute error */
        *total_error += (outputs[index][i][j]-out_act[i][j])*
            (outputs[index][i][j]-out_act[i][j]);
    };
};

/*.....*/

void o_h_pass(outputs,index_h_act,hidden_delta,out_act,h_o,
    learning,hidden_delta)
/* Backward pass from output to hidden layers, compute derivatives pertaining
to the weights and update momentum in h_o[][1], also compute
double derivatives pertaining to each weight and store in h_o[][2] */

float outputs[NUM_SETS][PPS][NOU]; /* Desired outputs array */
int index; /* Index to current batch */
float h_act[PPS][NHU+1]; /* Hidden activations */
float out_act[PPS][NOU]; /* Outputs of network for batch */
float h_o[NOU][NHU+1][3]; /* Hidden to output weights */
float learning[3]; /* Learning rates and momentum */
float hidden_delta[PPS][NHU+1]; /* Temporary variable to store intermediate
result to backpropagate error to update weights
for input to hidden layer weights */
float hidden_ddelta[PPS][NHU+1]; /* Temporary variable to store intermediate
results for OBD, to be backpropagated for
computing 2nd derivatives for i_h layer */

{
    int i,j,k;
    float sum1,sum2;
    float output_delta[PPS][NOU]; /* Temporary variable, accounting for error and
        corresponding derivative, for each output
        unit, used to compute derivatives */
    float output_ddelta[PPS][NOU]; /* Temporary OBD array */

    /* Zero values to be backpropagated */
    for (i=0;i<PPS;i++)
    {
        for (j=0;j<NHU;j++)
        {
            hidden_delta[i][j]=0.0;
            hidden_ddelta[i][j]=0.0;
        };
    };

    /* Loop over all batch entries and outputs and compute error-related term */
    for (i=0;i<PPS;i++)
    {
        for (j=0;j<NOU;j++)
        {
            output_delta[i][j]=(outputs[index][i][j]-out_act[i][j])*
                (out_act[i][j]*(1.0-out_act[i][j]));
            output_ddelta[i][j]=output_delta[i][j]*(1.0-2.0*out_act[i][j])-
                (1.0-out_act[i][j])*(1.0-out_act[i][j])*
                out_act[i][j]*out_act[i][j];
        };
    };

    /* Loop over all output neurons */
    for (i=0;i<NOU;i++)
    {
        /* Loop over all hidden neurons connecting to the output neuron */
        for (j=0;j<NHU;j++)
        {
            /* Zero sums */
            sum1=0.0;
            sum2=0.0;

```

```

/* Loop over all entries in training batch */
for (k=0;k<PPS;k++)
{
    /* Update change */
    sum1 = sum1+(output_delta[k][i]*h_act[k][j]); /* First derivative */
    sum2 = sum2+(output_delta[k][i]*h_act[k][j]*h_act[k][j]); /* 2nd derivative */

    /* Update quantity to be backpropagated for input-hidden layer weights */
    hidden_delta[k][j] += (output_delta[k][i]*h_o(i)[j][0]);
    hidden_ddelta[k][j] += (output_delta[k][i]*h_o(i)[j][0]*h_o(i)[j][0]);
};

/* Update weights according to change over all training */
h_o(i)[j][0] += h_o(i)[j][1] + learning[1]*sum1;
/* Update momentum related term */
h_o(i)[j][1] = learning[2]*(h_o(i)[j][1]+sum1);
/* Store double derivative */
h_o(i)[j][2] += sum2;
};
};

);

/*****/

void h_i_pass(inputs_index,hidden_delta,hidden_ddelta,
             h_act,out_act,learning,i_h)
/* Backward pass from hidden to input layers, compute derivatives pertaining
to the weights and store momentum update in i_h[][][1] */

float inputs[NUM_SETS][PPS][NIU+1]; /* Input training vectors */
int index; /* Index to current batch */
float hidden_delta[PPS][NHU+1]; /* Backpropagated error quantity from
hidden-output layer */
float hidden_ddelta[PPS][NHU+1]; /* Backpropagated double derivative quantity
from input to hidden layer */
float h_act[PPS][NHU+1]; /* Hidden activation for current batch */
float i_h[NHU][NIU+1][3]; /* Weights from input to hidden layer */
float learning[3]; /* Learning rates and momentum */

{
    int i,j,k;
    float sum1,sum2;
    float h_delta[PPS][NHU]; /* Temporary variable for single derivative */
    float h_ddelta[PPS][NHU]; /* Temporary variable for double derivative */

    /* Using backpropagated quantities, scale with derivatives to generate other
quantities to be used in generating derivatives, corresponding to each hidden
neuron */

    for (i=0;i<PPS;i++)
    {
        for (j=0;j<NHU;j++)
        {
            h_delta[i][j]=(hidden_delta[i][j])*(h_act[i][j])*(1.0-h_act[i][j]);
            h_ddelta[i][j]=h_delta[i][j]*(1.0-2.0*h_act[i][j]) +
                hidden_ddelta[i][j]*(1.0-h_act[i][j])*(1.0-h_act[i][j])
                h_act[i][j] * h_act[i][j];
        };
    };

    /* Loop over all hidden neurons */
    for (i=0;i<NHU;i++)
    {
        /* Loop over all input neurons */
        for (j=0;j<NIU;j++)
        {
            /* Zero sums */
            sum1=0.0;
            sum2=0.0;
            for (k=0;k<PPS;k++)
            {
                /* Update sums */
                sum1=sum1+(h_delta[k][i]*inputs[index][k][j]); /* Single derivative */
                sum2=sum2+(h_delta[k][i]*inputs[index][k][j]*inputs[index][k][j]); /* 2nd
derivative */
            };
            /* Update weight */
            i_h[i][j][0] += i_h[i][j][1] + learning[0]*sum1;

```

```

/* Update momentum term */
i_h[i][j][1] = learning[2]*(i_h[i][j][1]+sum1);
/* Store double derivative */
i_h[i][j][2] += sum2;
};
};

/******
void go(arglist) /* Routine to start backpropagation routine, in response
to user typing "go <n>" at BP2OBD prompt. Will go n
iterations else if n not specified, will loop until
stopped by user */
char *arglist;
{
char c;
int m,n;
int counting_epochs=0;
int run_epoch_count=0;

/* Check if number of iterations specified (n) */
if (arglist!=NULL && strlen(arglist)>0)
if (sscanf(arglist,"%d",&run_epoch_count)==1)
counting_epochs=1;
if (counting_epochs==1 & run_epoch_count<=0)
return;

/* Start Iteration-Error display */
printf(" Epoch      Average Error per Output Unit\n");
while(1) {
learning[0]=lrc1;
learning[1]=lrc2;
learning[2]=momentum;

/* Zero iteration error */
total_error=0.0;

/* Loop over all batch of size PPS */
for (set=0;set<NUM_SETS;set++)
{
/* Input to hidden pass for set batch */
i_h_pass(inputs,set,i_h_act);
/* Hidden to output pass */
h_o_pass(h_act,h_o_out_act);
/* Calculate error for batch */
calc_error(outputs,set,out_act,&total_error);
/* Backward pass from output to hidden, update weights in hidden-output layer
and compute double derivatives */
o_h_pass(outputs,set,h_act,hidden_delta,out_act,h_o_learning,hidden_delta);
/* Backward pass from hidden to input, update weights in input-hidden layer
and compute double derivatives */
h_i_pass(inputs,set,hidden_delta,hidden_delta,h_act,out_act,learning,i_h);
};

/* Increment iteration count */
epoch_count=epoch_count+1;

/* Print error */
total_error/= error_scale;
printf("%d      %f\n",epoch_count,total_error);

/* Check if done n iterations */
if(counting_epochs)
{
run_epoch_count -= 1;
if(run_epoch_count<=0)
{
return;
};
};

/* Zero double derivatives for next iteration */
for (m=0;m<NHU;m++)
{
for(n=0;n<NIU;n++)
{
i_h[m][n][2]=0.0;
};
};
}

```

```

for (m=0;m<NOU;m++)
{
for(n=0;n<NHU;n++)
{
h_o[m][n][2]=0.0;
};
};
};

/*****
int read_comlin(argc, argv) /* Read command line input, and parse */
int argc;
char *argv[];
{
double atof();
int i;

/* Load default values, overwrite later with user specified values if given*/
strcpy(input_pattern_file, INPUT_PATTERN_FILE);
strcpy(output_pattern_file, OUTPUT_PATTERN_FILE);
strcpy(weight_file, WEIGHT_FILE);
strcpy(saved_weight_file, SAVE_FILE_INIT);
lrc1 = 0.1;
lrc2 = 0.1;
momentum = 0.5;
thresh = 0.0001;
decay = 0.8;

/* Parse input and overwrite corresponding default settings */
for(j=1; j<argc; j++)
if(argv[j][0]!='-')
switch(argv[j][1]){
case 'i':
j++;
strcpy(input_pattern_file, argv[j]);
break;
case 'o':
j++;
strcpy(output_pattern_file, argv[j]);
break;
case 'w':
j++;
strcpy(weight_file, argv[j]);
break;
case 's':
j++;
strcpy(saved_weight_file, argv[j]);
break;
case 'l':
lrc1 = atof(argv[j+1]);
lrc2 = atof(argv[j+1]);
break;
case 'm':
momentum = atof(argv[j+1]);
break;
default:
printf("Usage: host [-l learn1 learn2] [-m momentum]");
printf(" [-i input_patn_file] [-o output_patn_file] ");
printf("[-w initial_weight_file] [-s saved_weight_file]\n");
return(-1);
}
return(0);
}

/*****
int initialize () /* Read weights, read input and desired training files */
{
if(read_initial_weights())
return(-1);
if(read_patterns())
return(-1);
return(0);
}

/*****
void write_weight_file(arglist) /* Saves weight file */
char *arglist;

```

```

{
    char filename[256];
    FILE *fd;
    int iu, hu, ou;

    /* Open file to write to, if not possible, exit */
    str_arg(arglist, "weights filename: ", filename);
    if ((fd = fopen(filename, "w")) == NULL) {
        fprintf(stderr, "could not open %s\n", filename);
        return;
    }

    /* Loop through each layer, input to hidden first and hidden to output and
    write out...see header at start of program for format */
    for(hu=0; hu<NHU; hu++){
        for(iu=0; iu<NIU+1; iu++){
            fprintf(fd, "%f\n", i_h[hu][iu][0]);
        }

        for(ou=0; ou<NOU; ou++){
            for(hu=0; hu<NHU+1; hu++){
                fprintf(fd, "%f\n", h_o[ou][hu][0]);
            }
        }

    /* Write out number of iterations */
    fprintf(fd, "%d epochs\n", epoch);
    printf(" written\n");
    fclose(fd);
}
/*****
void write_dd_file(arglist) /* Write out double derivatives for each weight */
char *arglist;

{
    char filename[256];
    FILE *fd;
    int iu, hu, ou;

    /* Open file to write to, if not possible, exit */
    str_arg(arglist, "dd filename: ", filename);
    if ((fd = fopen(filename, "w")) == NULL) {
        fprintf(stderr, "could not open %s\n", filename);
        return;
    }

    /* Loop through each layer, input to hidden first and hidden to output and
    write out double derivatives...see header at start of program for format */
    for(hu=0; hu<NHU; hu++){
        for(iu=0; iu<NIU+1; iu++){
            fprintf(fd, "%f\n", i_h[hu][iu][2]);
        }

        for(ou=0; ou<NOU; ou++){
            for(hu=0; hu<NHU+1; hu++){
                fprintf(fd, "%f\n", h_o[ou][hu][2]);
            }
        }

    /* Write out number of iterations */
    fprintf(fd, "%d epochs\n", epoch);
    printf(" written\n");
    fclose(fd);
}
/*****
void write_saliency_file(arglist)
/* Write out double derivatives for each weight */
char *arglist;

{
    char filename[256];
    FILE *fd;
    int iu, hu, ou;
    float temp;

    /* Open file to write to, if not possible, exit */
    str_arg(arglist, "dd filename: ", filename);
    if ((fd = fopen(filename, "w")) == NULL) {
        fprintf(stderr, "could not open %s\n", filename);
        return;
    }
}

```

```

/* Loop through each layer, input to hidden first and hidden to output and
write out double derivatives...see header at start of program for format */
for(hu=0;hu<NHU;hu++){
  for(iu=0;iu<NIU;iu++){
    temp=0.5*i_h[hu][iu][0]*i_h[hu][iu][0]*i_h[hu][iu][2];
    fprintf(fd, "%f\n", temp);
  }

  for(ou=0;ou<NOU;ou++){
    for(hu=0;hu<NHU+1;hu++){
      temp=0.5*h_o[ou][hu][0]*h_o[ou][hu][0]*h_o[ou][hu][2];
      fprintf(fd, "%f\n", temp);
    }
  }

/* Write out number of iterations */
fprintf(fd, "%d epochs\n", epoch);
printf(" written\n");
fclose(fd);
}
/*****
void odd() /* Perform optimal brain damage, one iteration must have been done
to make this run */
{
  float wt;
  float min,max; /* Maximum and minimum saliency values */
  float temp;
  int i,j;
  float cutoff; /* Threshold x Maximum Saliency */
  int n,c;
  extern double fabs();

  max=-1.0E+31;
  min=-max;

/* Find maximum saliency over all layers */
  for (i=0;i<NHU;i++)
    for(j=0;j<NIU;j++){
      {
        wt=i_h[i][j][0];
        temp=fabs(wt*i_h[i][j][2]*i_h[i][j][2]);
        if (max<temp) max=temp;
        if (min>temp) min=temp;
      };

      for (i=0;i<NOU;i++)
        for(j=0;j<NHU;j++){
          {
            wt=h_o[i][j][0];
            temp=fabs(wt*h_o[i][j][2]*h_o[i][j][2]);
            if (max<temp) max=temp;
            if (min>temp) min=temp;
          };
        }

/* Compute threshold for deletion, ratio of maximum */
      cutoff=thresh*max;
      printf("Maximum: %f Minimum: %f\n",max,min);
      c=0;
      n=0;

/* Loop through all weights, decay those below cutoff */
      for (i=0;i<NHU;i++)
        for(j=0;j<NIU;j++){
          {
            n++;
            temp=i_h[i][j][0]*i_h[i][j][2]*i_h[i][j][2];
            if( fabs(temp) < cutoff )
              {
                i_h[i][j][0] *= decay;
              };
            c++;
            i_h[i][j][2]=0.0;
          };
        }

      for (i=0;i<NOU;i++)
        for(j=0;j<NHU;j++){
          {
            n++;
            temp=h_o[i][j][0]*h_o[i][j][2]*h_o[i][j][2];
            if( fabs(temp) < cutoff )
              {

```



```

        h_o[i][j][0] *= decay;
        c++;
    };
    h_o[i][j][2]=0.0;
};

printf("Changed %d weights (%g percentage)\n",c,((float)c/n)*100.0);

};
/*****
int read_initial_weights() /* Reads initial weights */
{
    int iu, hu, ou;
    FILE *fp;

    /* Open initial weights file */
    if((fp = fopen(weight_file,"r")) == NULL){
        printf("cannot open %s\n", weight_file);
        return(-1);
    }

    /* Read in weights, format specified in header */
    printf("Reading weights ...");
    for(hu=0; hu<NHU; hu++){
        for(iu=0; iu<(NIU+1); iu++){
            fscanf(fp,"%f",&i_h[hu][iu][0]);
            i_h[hu][iu][1] = 0.0;
        };

        for(ou=0; ou<NOU; ou++){
            for(hu=0; hu<NHU+1; hu++){
                fscanf(fp,"%f",&h_o[ou][hu][0]);
                h_o[ou][hu][1] = 0.0;
            };
        };
    }

    /* Read in number of iterations done with these weights */
    fscanf(fp,"%d epoch\n",&epoch);
    fclose(fp);
    printf(" read\n");
    return(0);
}
/*****
void print_nn_constants() /* Print header information to screen, summarizing
    input and details of network */
{
    printf("Starting fully connected NN with 1 hidden layer\n");
    printf("%3d Input units (w/o bias unit)\n", NIU);
    printf("%3d Hidden units (w/o bias unit)\n", NHU);
    printf("%3d Output units\n", NOU);
    printf("\nLearning Rates %6.3f %6.3f\n", lrc1, lrc2);
    printf("Momentum %6.3f\n", momentum);
    printf("\nTraining Files\n");
    printf("Input Pattern %s\n", input_pattern_file);
    printf("Output Pattern %s\n", output_pattern_file);
    printf("Weight %s\n", weight_file);
}
/*****
void str_arg(string, prompt, name) /* String to argument conversion */
char *string, *prompt, *name;
{
    if (string==NULL || sscanf(string,"%s",name)!=1) {
        printf(prompt);
        scanf("%s",name);
    };
}
/*****
void exit_program(arglist) /* In response to "quit" at BP2OBD prompt */
char *arglist;
{
    char answer[12];

    /* Check if weights to be saved, if so, save them and exit */

    str_arg(arglist, "save weights file [y/n]: ", answer);
    if(answer[0]!='n' && answer[0]!='N')
        write_weight_file(save_file);
    exit();
}
/*****
int cmdsearch( str, rest ) /* Search for command number given input string

```

```

        entered by user. Return arguments in rest */
char *str; /* Input string */
char *rest; /* Arguments */
{
    int i, l;

    /* Loop over all command entries */
    for (i=0; i<cmdtblsize; i++)
    {
        l = strlen( cmdtbl[i].cmdstr );
        /* If match found, return number, and arguments */
        if (!memcmp(str, cmdtbl[i].cmdstr, l))
        {
            strcpy( rest, str + l );
            return (cmdtbl[i].cmdno);
        }
    }
    strcpy( rest, str );
    return (CMD_NULL);
}
...../

```

```

/*
 * -----
 *      file:      cparser2obd.h
 *      desc:      simple command parser
 *      by:        patrick ko
 *      date:      22 aug 91
 * -----
 */

#define      CMD_NULL 0

typedef      struct      {
    int      cmdno;
    char      *cmdstr;
}      CMDTBL;

/*
 *      #define your own commands here starting from 1
 *      ..* modifiable *.
 */

#define      read_weights      1
#define      write_weights      2
#define      write_dd      3
#define      write_saliency      4
#define      obdrum      5
#define      goiter      6
#define      quit      7
#define      lrca      8
#define      lrob      9
#define      momentumc      10
#define      threshold      11
#define      decayconstant      12
#define      epochnum      13

```

```

PROGRAM: FWDBP2
Classify patterns using 1 Hidden Layer Fully Connected NN

```

```

USAGE:
fwdbp2 [-w weight_file] [-i input_patt_file] [-o output_pattern_file]

```

```

OPTIONS:
-w weight_file      weight file
-i input_patt_file  file with input patterns to be classified
-o output_patt_file file to which the output pattern written to

```

```

DEFAULTS:
weight_file      specified in "constants.h"
input_patt_file  standard input
output_patt_file standard output

```

```

NOTE:
1. The network parameters are specified in the file "constants.h"
2. The program must be recompiled for new network architecture.
3. The directory for sigmoid data is specified in SIGMOID_DIR

```

```

AUTHOR:
Rajesh Jha    July 31 1991

```

```

#include <math.h>
#include <stdio.h>
#include "constants.h"

#define PROGNAME "fwdbp2"
#define OUTPUT_FMT "%f"

float i_h[NHU][NIU+1]; /* input-to-hidden weights */
float h_o[NOU][NHU+1]; /* hidde-to-output weights */

char weight_file[256];
char input_patt_file[256];
char result_file[256];
FILE *infile, *outfile;
int dbgfl;

/*-----*/
main(argc, argv)
int argc;
char *argv[];
{
    if(read_comlin(argc, argv))
        exit(0);
    if(read_weights())
        exit(-1);
    if(dbgfl){
        write_preamble();
        write_weights();
    }
    while(!classify());
    exit(0);
}

/*-----*/
classify()
{
    float input[NIU+1];
    float hidden[NHU+1];
    float output[NOU];
    int i, j, k;
    float temp;

    if(read_in patt(input, NIU+1)){
        fprintf(stderr, "%s: patterns in %s classified\n", PROGNAME, input_patt_file);
        return(-1);
    }
    if(dbgfl)
        write_in patt(input, NIU+1); /* input to hidden */
    for(i=0; i<NHU; i++){
        temp = 0.0;
        for(j=0; j<NIU; j++)

```

```

        temp = temp + i_h[i][j] * input[j];
        hidden[i] = 1.0 / (1.0 + exp(-1.0 * temp));
    }
    hidden[NHU] = 1.0; /* hidden to output */
    for (i=0; i<NOU; i++){
        temp = 0.0;
        for (j=0; j<=NHU; j++){
            temp = temp + h_o[i][j] * hidden[j];
        }
        output[i] = 1.0 / (1.0 + exp(-1.0 * temp));
    }

    write_output(output, NOU);
    return(0);
}

/*-----*/
read_input(fvector, n)
float *fvector;
int n;
{
    int i;

    fvector[0] = 1.0; /* bias unit */
    for(i=1; i<n; i++){
        if(fscanf(infile, "%f", &fvector[i]) == EOF)
            return(-1);
    }
    return(0);
}

write_input(fvector, n)
float *fvector;
int n;
{
    int i;

    for(i=1; i<n; i++){
        fprintf(outfile, "%g ", fvector[i]);
        fprintf(outfile, "\n");
    }
}

/*-----*/
write_output(fvector, n)
float *fvector;
int n;
{
    int i;

    for(i=0; i<n; i++){
        fprintf(outfile, OUTPUT_FMT, fvector[i]);
        fprintf(outfile, "\n");
    }
}

/*-----*/
write_preamble()
{
    fprintf(outfile, "  Input Units %d\n", NIU);
    fprintf(outfile, "  Hidden Units %d\n", NHU);
    fprintf(outfile, "  Output Units %d\n", NOU);
    fprintf(outfile, "  Fully Connected\n");
    fprintf(outfile, "  Weight File %s\n", weight_file);
    fprintf(outfile, "\nClassifying\n\n");
}

/*-----*/
int
read_weights()
{
    int iu, hu, ou;
    FILE *fp;
    int epoch;

    if(((fp = fopen(weight_file, "r")) == NULL)){

```

```
break;
default:
    print_usage();
    return(-1);
}
return(0);
}

print_usage()
{
    fprintf(stderr, "Usage: %s", PROGRAMNAME);
    fprintf(stderr, " [-w weight_file] [-i input_path_file] [-o result_file] ");
    fprintf(stderr, "\n");
}
```

```

fprintf(stderr, "%s cannot open %s\n", PROGRAMNAME, weight_file);
return(-1);
}

```

```

if(debug)
fprintf(stderr, "Reading weights from %s ... ", weight_file);
for(hu=0; hu<NHU; hu++)
for(iu=0; iu<(NIU+1); iu++)
fscanf(fp, "%f", &i_h[hu][iu]);

for(ou=0; ou<NOU; ou++)
for(hu=0; hu<NHU+1; hu++)
fscanf(fp, "%f", &h_o[ou][hu]);

fscanf(fp, "%d epochs\n", &epoch);
if(debug)
fprintf(stderr, " read\n");
fclose(fp);
return(0);
}

```

```

/*-----*/
write_weights()

```

```

{
int iu, hu, ou;

for(hu=0; hu<NHU; hu++)
for(iu=0; iu<(NIU+1); iu++){
fprintf(outfile, "%g ", i_h[hu][iu]);
fprintf(outfile, "\n");
}
for(ou=0; ou<NOU; ou++)
for(hu=0; hu<NHU+1; hu++){
fprintf(outfile, "%g ", h_o[ou][hu]);
fprintf(outfile, "\n");
}
}

```

```

/*-----*/

```

```

int
read_comlin(argc, argv)
int argc;
char *argv[];
{
int i;

debug = 0;
strcpy(weight_file, WEIGHT_FILE);
infile = stdin;
outfile = stdout;
strcpy(input_patt_file, "stdin");
strcpy(result_file, "stdout");

for(i=1; i<argc; i++)
if(argv[i][0]!='-'){
switch(argv[i][1]){
case 'i':
i++;
strcpy(input_patt_file, argv[i]);
if((infile = fopen(argv[i], "r")) == NULL){
fprintf(stderr, "%s cannot open %s\n", PROGRAMNAME, argv[i]);
return(-1);
}
break;
case 'o':
i++;
strcpy(result_file, argv[i]);
if((outfile = fopen(argv[i], "w")) == NULL){
fprintf(stderr, "%s cannot open %s\n", PROGRAMNAME, argv[i]);
return(-1);
}
break;
case 'w':
i++;
strcpy(weight_file, argv[i]);
break;
case 'D':
debug = 1;
}
}
}

```


UNCLASSIFIED

SECURITY CLASSIFICATION OF FORM
(highest classification of Title, Abstract, Keywords)

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Establishment sponsoring a contractor's report, or tasking agency, are entered in section 8.)

MCMASTER UNIVERSITY
COMMUNICATIONS RESEARCH LABORATORY
1280 MAIN STREET WEST, HAMILTON, ONTARIO L8S 4K1

2. SECURITY CLASSIFICATION

(overall security classification of the document including special warning terms if applicable)

UNCLASSIFIED

3. TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C or U) in parentheses after the title.)

AUTOMATIC MODULATION RECOGNITION BASED ON SPATIAL PATTERN CLASSIFICATION USING
ARTIFICIAL NEURAL NETWORKS (U)

4. AUTHORS (Last name, first name, middle initial)

GHANI, NASIR

5. DATE OF PUBLICATION (month and year of publication of document)

28 JULY 1992

6a. NO. OF PAGES (total containing information. Include Annexes, Appendices, etc.)
193

6b. NO. OF REFS (total cited in document)
29

7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

FINAL CONTRACTOR REPORT

8. SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include the address.)

DEFENCE RESEARCH ESTABLISHMENT OTTAWA
NATIONAL DEFENCE
SHIRLEY BAY, OTTAWA, ONTARIO K1A 0K2 CANADA

9a. PROJECT OR GRANT NO. (if appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant)

PROJECT 041LK

9b. CONTRACT NO. (if appropriate, the applicable number under which the document was written)

W7714-1-9531/01-ST

10a. ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique to this document.)

10b. OTHER DOCUMENT NOS. (Any other numbers which may be assigned this document either by the originator or by the sponsor)

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

☒ Unlimited distribution

☐ Distribution limited to defence departments and defence contractors; further distribution only as approved

☐ Distribution limited to defence departments and Canadian defence contractors; further distribution only as approved

☐ Distribution limited to government departments and agencies; further distribution only as approved

☐ Distribution limited to defence departments; further distribution only as approved

☐ Other (please specify):

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected.)

UNCLASSIFIED

SECURITY CLASSIFICATION OF FORM

DCD03 2/06/87

UNCLASSIFIED

SECURITY CLASSIFICATION OF FORM

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

(U) This research involves work on an electronic countermeasures contract for the Defence Research Establishment Ottawa (DREO). In this project, an automatic modulation recognition system has been designed and tested based on backpropagation neural networks. This system successfully distinguishes between the specified 10 signal classes to a very high degree and uses a Welch periodogram preprocessor. Experimental results show that the neural networks match and even outdo the performance of the conventional K-nearest neighbour classifier for the same preprocessor, upper 90's percentages. Significant optimization of the neural networks is also shown using the optimal brain damage pruning algorithm.

(U) ^{discusses} The design issues, preprocessing techniques, backpropagation algorithms, implementation details, ^{and} results of this project are fully discussed in Part I of this document. ^{9th} Part II discusses the specifics of the software developed during the project. //

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

MODULATION RECOGNITION
AUTOMATIC MODULATION CLASSIFICATION
SIGNAL DETECTION
SIGNAL IDENTIFICATION
COMMUNICATION SIGNALS

UNCLASSIFIED

SECURITY CLASSIFICATION OF FORM

